VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Generování map a struktur pro projekt Jiný Kosmos

Map and Structures Generation for Project Other Cosmos

2017 Jakub Brecher

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Brecher**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Generování map a struktur pro projekt Jiný Kosmos
Map and Structures Generation for Projedct Other Cosmos

Jazyk vypracování: čeština

Zásady pro vypracování:

Práce přímo navazuje na vývoj simulačního prostředí Jiný Kosmos, ketrý probíhá v rámci semestrálních projektů. Cílem práce je vytvořit generátory prostředí pro projekt Jiný Kosmos. Především generátory 3D krajiny a krajinných útvarů (hor, řek, jezer, pláží, jeskynní, ...). Tento generátor bude úzce spolupracovat s generátorem biotopů, který bude respektovat konfiguraci generované planety a jejich klimatických podmínek. Dalším typem generátoru je pak generátor rozsáhlejších struktur, jako jsou stavby (bludiště, domy, hrady, ...) tento generátor umožní buď plně automatické generování nebo generování spojováním předem vytvořených celků nebo kombinace obou přístupů.

Generátory umožní:
1. Generování biotopů, který budou respektovat konfiguraci generované planety a jejich klimatických podmínek.
2. Generování 3D krajiny a krajinných útvarů (hor, řek, jezer, pláží, jeskynní, ...).
3. Automatické generování 3D struktur (bludiště, domy, hrady, ...).
4. Generování větších struktur z předem připravených celků (mistnosti, schodiště, ...).
5. Vkládání 3D modelů postav a jejich chování do vygenerovaného světa.

Práce bude obsahovat:
1. Přehled používaných metod a algoritmů.
2. Implementaci výše popsané funkcionality.
3. Experimenty a porovnání výsledků různých přístupů při generování.
4. Programátorskou dokumentaci řešení s využitím diagramů jazyka UML.

Seznam doporučené odborné literatury:

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí vzorů. Grada. Praha 2003. ISBN 8024703025
[2] Polygonal Map Generation for Games. Amit Patel's Home Page [online]. Standford [cit. 2016-03-07]. Dostupné z: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/

Dále dle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

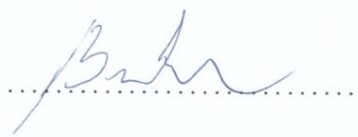
prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 28. dubna 2017



Rád bych na tomto místě poděkoval Ing. Davidu Ježkovi, Ph.D. za cenné rady při tvoření této diplomové práce

Abstrakt

Cílem práce je rozšířit simulační prostředí Jiný Kosmos o procedurální generování světa. Především se jedná o procedurální generování 3D krajiny a krajinných útvarů (hor, řek, jezer, pláží, jeskyní,...). Dále pak o generaci biotopů, jež určují klimatické podmínky dané oblasti, které pak následně ovlivňují typ a povrch krajiny. S tím vším úzce souvisí generátory stromů, budov a řek. Uživatel si může sám upravit podmínky generací a tím tak upravit vzhled Kosmosu podle svých představ. Aplikace je implementována v jazyce Java. Pro podporu 3D vykreslování a fyziky je použitý rámec LibGDX, díky tomu se tak hráč může po světě libovolně procházet. V prvních částech se věnuji popisu algoritmů pro procedurální generování světa a následně jejich implementaci. Dále pak popisu použitých technologií a samotné implementaci aplikace.

Klíčová slova: Java, LibGDX, procedurální generování, Simplex noise, Voxel, Minecraft, simulační prostředí, Voroného diagram, biomy, generování stromů, generování řek, generování budov, 3D vykreslování, Bullet, Gradle

Abstract

The goal of the thesis is to enrich the simulation environment Jiný Kosmos with a procedural generation of the world, mainly generation of 3D landscapes (mountains, rivers, lakes, beaches, etc.). Next, to implement the generation of biotopes that determine the climatic conditions of the given area, which subsequently affect the type and the surface of the landscape. Everything is closely linked to the generators of trees, buildings and rivers. The user can modify the conditions of the generation by himself and thus modify the Kosmos as he wants. The application is implemented in Java and LibGDX framework, which is used to support 3D rendering and physics, allowing the player to move and navigate around the world freely. In the first part, I describe the landscape generation algorithms and consequently their implementation. Furthermore, I cover the descriptions of the technologies used and the implementation of them.

Key Words: Java, LibGDX, procedural generation, Simplex noise, Voxel, Minecraft, simulation environment, Voronoi diagram, biomes, tree generation, river generation, building generation, 3D render, Bullet, Gradle

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
FBX	–	Formát 3D modelu
FPS	–	Frames Per Second
GUI	–	Graphical User Interface
OpenGL	–	Open Graphics Library
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato práce je součástí projektu „Jiný Kosmos“. Jedná se o 3D simulační prostředí podobné známe hře Minecraft [1]. Hlavním prvkem jsou tedy tzv. „bloky“, ze kterých je složen celý 3D svět. Tyto bloky lze různě odebírat a přidávat. Tímto způsobem následně generovat a přetvářet celou krajinu.

Hlavním cílem této práce je tedy vytvořit reálně vypadající krajinu obsahující větší struktury, jako např. stromy, jezera, řeky a budovy. K vytvoření světa je použité procedurální generování terénu pomocí několika různých algoritmů, které jsou popsány v kapitole 2.

Generování mapy je modifikovatelné a uživatel má tedy možnost volby jakým způsobem a s jakými parametry bude svět generován. Může si tak vytvořit svět přesně podle svých představ.

Součástí tohoto světa jsou také 3D modely postav, které jsou do světa zasazeny podle různých kritérií. Je zapotřebí, aby se tyto postavy uměly ve světě Kosmosu pohybovat. Aby pohyb vypadal přirozeně, je potřeba pro modely zajistit animace.

Projekt je implantován v jazyce Java a pro lepší a rychlejší práci je použitý rámec LibGDX. Aplikaci lze spustit na osobním počítači.

2 Procedurální generování světa

Náhodně vygenerovanou krajinu lze využít nejen u her, ale i v mnoha jiných odvětvích. Může se jednat o různé simulátory přírodních jevů, výukové programy, filmy atd. V poslední době se můžeme setkat se spousty nových her, ve kterých je využito právě procedurálního generování světa.

Ať už se jedná o plně nebo částečně náhodný svět, princip je vždy podobný. Nejdříve je potřeba vytvořit povrch krajiny. Toho lze docílit za pomocí šumových funkcí. Použitím základních 2D funkcí jako např. Perlin Noise, Random faults, Multifractals [2] jsme schopni vytvořit výškovou mapu a z ní následně vygenerovat terén světa. Problém nastává ve chvílí. kdy chceme v naší krajině mít převisy nebo jeskyně. Za pomocí 2D šumových funkcí nikdy nedosáhneme toho, že v určité výšce bude zem a pod ní prázdno (převis). Proto je potřeba sáhnout ke komplexnějším funkcím jako je například Simplex Noise [3].

Poté co máme vytvořený terén je potřeba jej nějakým způsobem obarvit. Můžeme se inspirovat naší planetou, která je rozdělná do několika biomů a stejně tak rozdělit i náš nově vytvořený svět. Dalším možným způsobem je rozdělit různé povrchy podle určitý parametrů např. Teplota, výška, vlhkost.

Dále můžeme svět obohatit o různé struktury jako například stromy, jezera, řeky a budovy. Dále pak můžeme do světa přidat živé bytosti a podle předem určených parametrů je vhodně rozmístit. Například se můžeme řídit naší přírodou a to tak, že v poušti se budou nacházet škorpioni a v lese zase vlci. Nebo si vymyslet naprosto vlastní rozdělení a i vlastní bytosti. Fantazii se zde meze nekladou.

2.1 Procedurální generování nejen ve světě Kosmos

Jak už jsem se zmínil, existuje spousta projektů, her, simulátorů. . . , jež využívají procedurálního generování obsahu. Na některé z nich se podíváme blíže.

2.1.1 No Man's Sky

Ve hře No man's sky z roku 2016 je procedurálně generováno téměř vše. Ať už se jedná o terén, stromy nebo žijící tvory vše je generováno procedurálně, dokonce i hudba.

Algoritmus skrývající se v jádře slibuje až 18 trilionu různých planet. Při takovém velkém čísle je celkem jasné, že některé planety nebudou od sebe příliš odlišné. Mohou se lišit například jen v detailech, ale i přesto je to úchvatné číslo. Základní vzhled planety se určuje podle několika základních podmínek jako např. Poloměr, procento vody, hustota zalesnění, teplota, zdroje a suroviny, barva povrchu a vody, atmosféra, toxicita, druhy rostlin a zvířat a několik set dalších méně významných podmínek. Ve světě No Man's Sky pak můžete narazit na spousty různorodých planet, některých kypřícím životem a jiných naprosto pustých a bez života.

Obrázek 1: Různé planety ze hry No Man's Sky [15]

Procedurální generování živočichů je zde samostatným komplexním odvětvím. Hned první den vydání této hry, bylo hráči objeveno přes 10 milionů druhů tvorů. To je více, než bylo objeveno na naší planetě. Každý tvor se skládá z různých částí těla (hlava, tělo, končetiny...). Z těchto jednotlivých částí je následně vytvořen unikátní tvor.

Obrázek 2: Tvorové ze hry No Man's Sky [16]

Stejně jako fauna je zde procedurálně generována i flora. Dělí se do několika základních skupin: houby, rostliny, stromy, kapradiny, korály, krápníkové a skalní. Dále můžeme narazit na rostliny s rozdílným chováním. Například masožravé rostliny neváhající na vás zaútočit atd...

Obrázek 3: Rostliny ze hry No Man's Sky

Hudba je zde generována za pomocí různých kombinací předem vytvořených audio stop v různých časových obdobích a situacích na základě mnoha různých pravidel v reálném čase.

2.1.2 Minecraft

Procedurální generace by se ale nikdy nedostala do tak vysokého postavení nebýt, až pohádkového úspěchu hry Minecraft. Ovšem jeho úspěch nestojí pouze na procedurálním generování světa. To samo o sobě stačit nemůže, málokoho by bavilo chodit po byť krásném světě, ale naprosto bezcílně. Minecraft tohle vynahrazujeme propracovaným systémem sběru surovin, ze kterých lze vytvořit různorodé předměty. Takto lze vytvořit i nové bloky použitelné jako stavební materiál pro hráčův úkryt. Ten totiž je potřeba, jelikož v noci na povrch vylézají různí nepřátelé s cílem zabít každého, kdo se v tomto světě vyskytuje.

Je zde několik set různých způsobů, jak efektivně zabít čas a je tedy jen na hráči, který si zvolí. Ať už bude tvořit bizarní stavby, objevovat nespočet různých lokací, potýkat se s ohromným počtem nepřátel, vyrábět lepší a lepší zbraně vždy a všude bude s ním přítomna ona procedurální generace světa.

Obrázek 4: Hráčem vytvořená stavba ze hry Minecraft [17]

Terén je zde generován za pomocí šumových funkcí jako například Perlin noise.

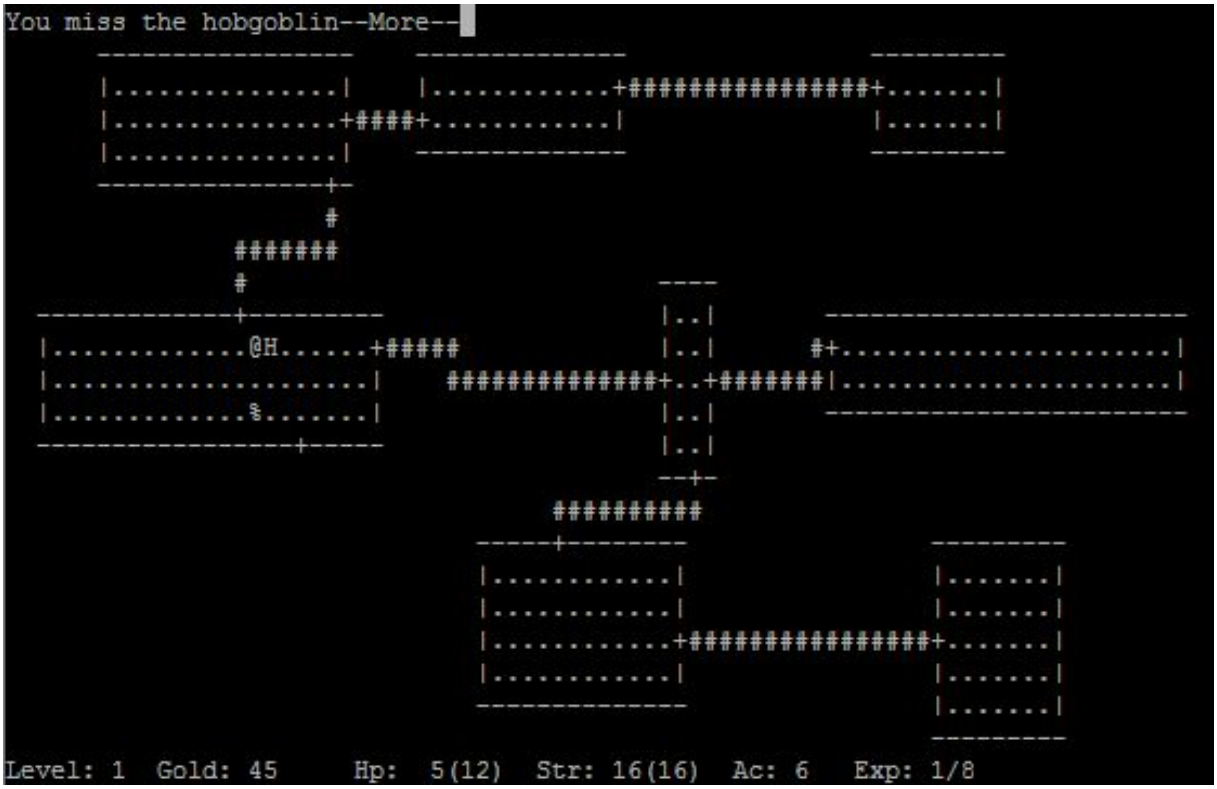
Obrázek 5: Terén ze hry Minecraft [18]

Pro generací jeskyní je použitý tzv. Perlin worms. Ten dokáže vytvořit úzké tunely, které jsou pak zakončeny větší jeskyní.

Obrázek 6: Jeskynní systém ze hry minecraft [19]

2.1.3 Starší tituly

Dnes se procedurální generace využívá zejména jako automatická výplň her pomocí matematických algoritmů, které se postarají například o stvoření světa, postav, předmětů atd... Díky tomu hra ideálně nebude nikdy stejná, obsah je dynamický. Ve výsledku to ulehčí práci vývojářům, jelikož nemusejí vše vytvářet ručně. Ve hře Rogue z roku 1980 bylo cílem proklestit si cestu na dno světa, který byl procedurálně generován. Důvod byl zde ale mnohem prostší a to nedostatek paměti pro uložení specifických map. Díky tomu se tato hra stala legendou.



Obrázek 7: Hra Rogue z roku 1980 [20]

Dalším titulem po Rogue, využívajícím procedurální generování ve větší míře, je slavná hra z roku 1984 Elite. Tehdejší osmibitové počítače nedokázali uchovat předem vytvořené galaxie, takže při každé hře byly náhodně vytvořeny. S procedurálním generováním se setkáme i u známé hry Diablo, kde jsou náhodně generovány předměty a jejich vlastnosti, stejně tak vlastnosti nepřátel. Na pár výjimek i většina lokací.

2.2 Terén

Pro vygenerování terénu se nejlépe hodí šumové funkce. S jejich pomocí jsme schopni vygenerovat výškovou mapu. Nejpoužívanější šumovou funkcí pro generaci terénu je Perlin noise a jeho další modifikace. Mezi další podobné funkce patří Random faults, Multifractals atd... Všechny tyto funkce jsem již popsal ve své bakalářské práci [2]. Tyto metody pro 3D prostor nestačí. Buď to nedokáži anebo jsou výpočetně příliš náročné.

2.2.1 Simplex noise

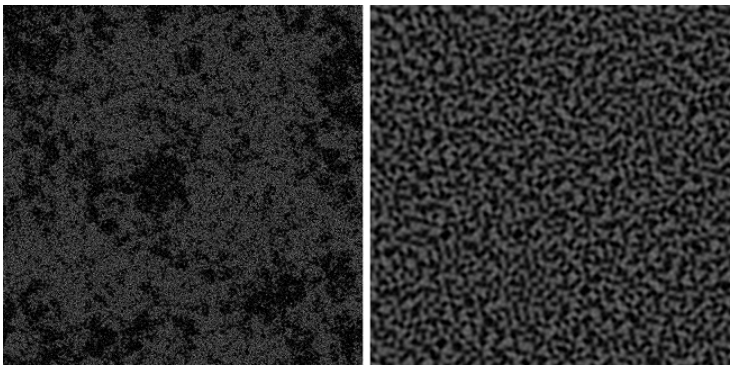
Ken Perlin v roce 2001 představil Simplex noise jako nástupce svého původního algoritmu Perlin noise. Klasický Perlin noise, díky kterému získal akademickou cenu, se v průběhu let stal hojně využívaným algoritmem. I přes své výhody měl několik nevýhod a omezení. Proto Ken Perlin navrhl tento nový algoritmus.

Některé z výhod Simplex noise:

- Má nižší výpočetní složitost
- Dokáže pracovat i ve vyšších dimenzích (3D, 4D a vyšší) a s mnohem menší výpočetní složitostí $O(N^2)$ oproti klasickému perlinu $O(2^N)$, kde N je počet dimenzí
- Nemá žádné znatelné směrové artefakty

Pro vygenerování šumové funkce je zapotřebí stanovit hodnoty základních parametrů:

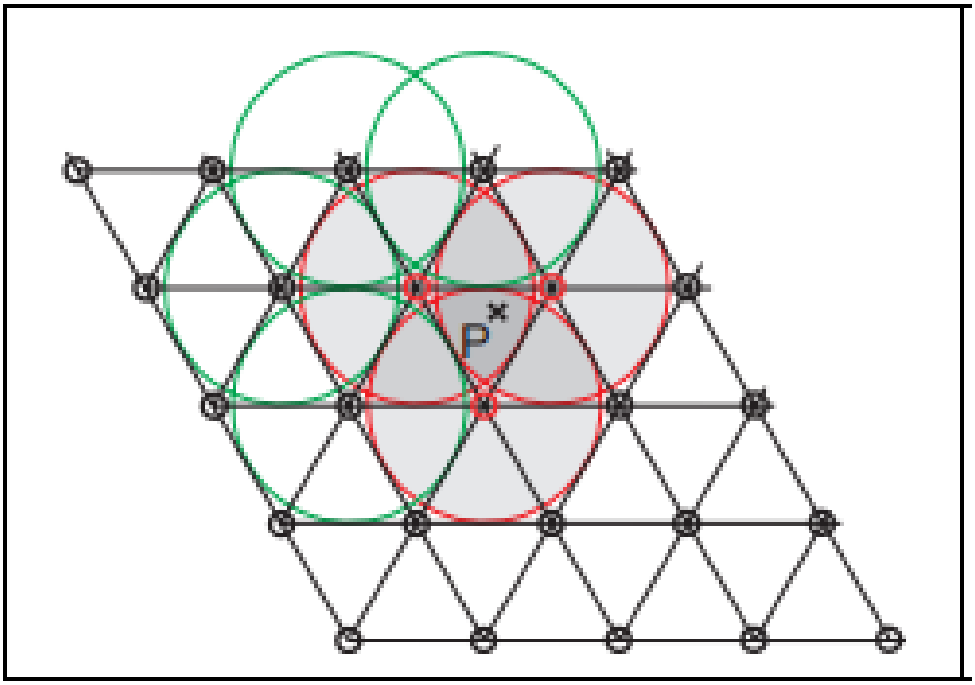
- **Amplituda** - Pomocí tohoto parametru určíme, jak rychle má probíhat přechod z nižších hodnot do vyšších a naopak. Jak je vidět na obrázku 8 s příliš vysokou amplitudou probíhá přechod hodnot mnohem rychleji.
- **Frekvence** - Určuje jak moc se mění hodnoty okolních bodů. Jak je vidět na obrázku 8 s příliš vysokou frekvenci mnohem více kolísá hodnota sousedních bodů.
- **Oktávy** - Počet oktáv udává, kolik šumových funkcí chceme spojit. S vyšší hodnotou jsou přechody hladší, ale narůstá tím složitost.



Obrázek 8: Vysoké hodnoty amplitudy (vlevo) a vysoké hodnoty frekvence (vpravo)

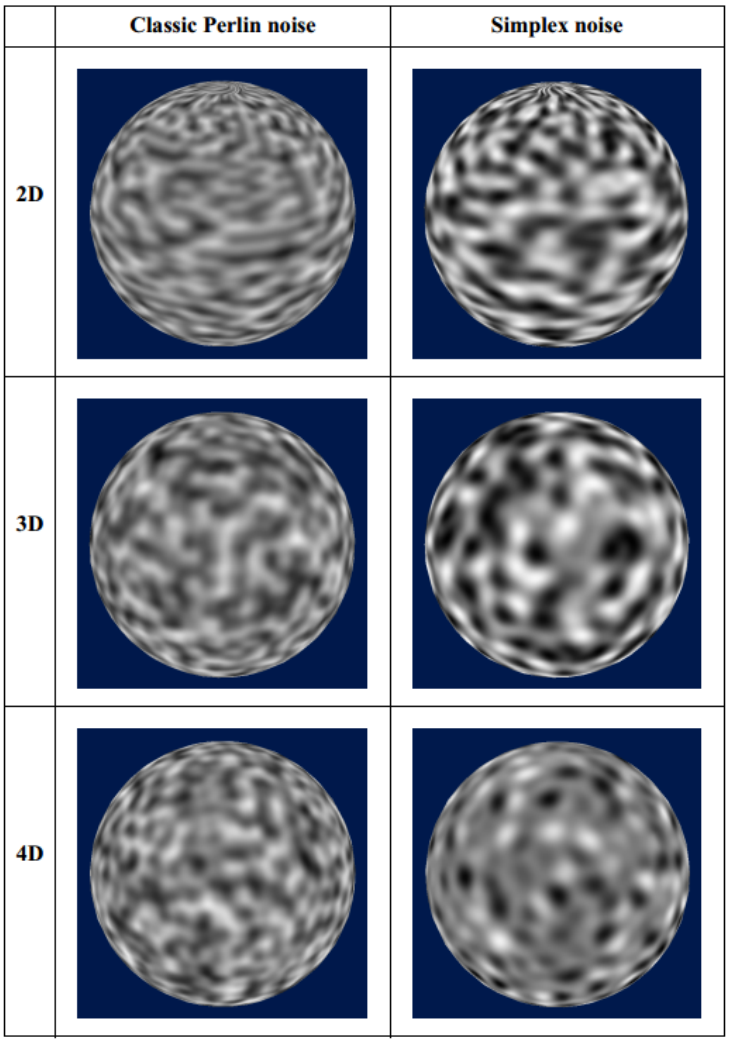
Základním problémem klasického Perlin noise je to, že podél každého rozměru zahrnuje po sobě následující interpolace. To znamená, že pro výpočet některého z bodu používá hodnoty předešlých interpolací, což výrazně zvyšuje výpočetní složitost, zejména pro vyšší dimenze. Simplex noise oproti tomu používá přímý součet hodnot z každého rohu. Kde hodnota je násobkem extrapolací a symetrické funkce radiálního útlumu. Radiální útlum je pečlivě zvolen tak, aby vliv od každého rohu dosáhl nuly před tím, než překročí hranici do dalšího „simplexu“. To znamená, že body uvnitř „simplexu“ budou ovlivněny pouze hodnotami z rohu konkrétního „simplexu“.

Na obrázku 9 bod P získává pro své hodnoty jen ze tří jader umístěných v okolních rozích (červené stínové kruhy). Hodnoty ze vzdálenějších jader (zelené kruhy) se rozpadnou na nuly ještě dřív, než překročí hranici do „simplexu“, ve kterém se nalézá bod P. To znamená, že hodnota šumu může být vypočítána jako součet pouze tři termínů.



Obrázek 9: Popis Simplex Noise [3]

Ve výsledku je tedy Simplex noise mnohem rychlejší, zejména pro vyšší dimenze. Nicméně, výsledný vizuální efekt je trochu odlišný od původní Perlin noise. Nedá se tedy využít jako „plug-in“ a rovnou jej nahradit za klasický Perlin noise pro projekty, které jej využívají. Výsledný efekt by totiž mohl být poněkud jiný. Liší se zejména ve vyšší dimenzích, jak vidíme na obrázku 10.



Obrázek 10: Rozdíl mezi Perlin noise a Simplex noise [3]

2.3 Biomy

Jedním se způsobů, jak obarvit terén, je rozdělit jej do několika různých biomů. Biom je dílčí oblastí biosféry a je charakterizován určitými klimatickými a půdními faktory, podle kterých je pak rozdělena fauna a flora. Stejně jako na naší planetě, tak i v našem nově vytvořeném světě můžeme na různých místech najít různé biomy. Podle toho o jaký biom se jedná, se zde nacházejí rostliny a zvířata. Můžeme tedy přidat pouště, zimní krajiny, lesy atd... Případně si vymyslet něco vlastního a originálního.

Poté co víme, jaké biomy budeme používat, je podstatné, všechny vhodně rozdělit. Pro co nejlepší efekt je potřeba, aby každý biom měl jinou velikost a tvar. Rozdělení světa podle různě velkých obdélníku asi nebude úplně nejvhodnější řešení a je tedy potřeba sáhnout po nějakém vhodném algoritmu.

2.3.1 Voroného diagram

K rozdělení biomu lze využit Voroného diagram. Jedná se o rovinný graf, jehož buňky jsou konvexní útvary. Vstupem je množina bodů P a výstupem je funkce $f : R^2 \to P$, která každému bodu x z R^2 přiřadí nejbližší bod z P.

K vytvoření Voroného diagramu lze pro měření vzdálenosti mezi dvěma body využít různých metod:

- Euklidovská vzdálenost - nejčastěji používanou metodou pro měření vzdálenosti. Jde o přímou vzdálenost mezi dvěma body. Je určena jako druhá odmocnina sumy čtvercových vzdáleností mezi pozicemi bodů. Je vždy větší nebo rovná nule. Euklidovská vzdálenost je speciálním případem Minkowského vzdálenosti s parametrem $\lambda = 2$.

  $$d_{ij} = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2}$$

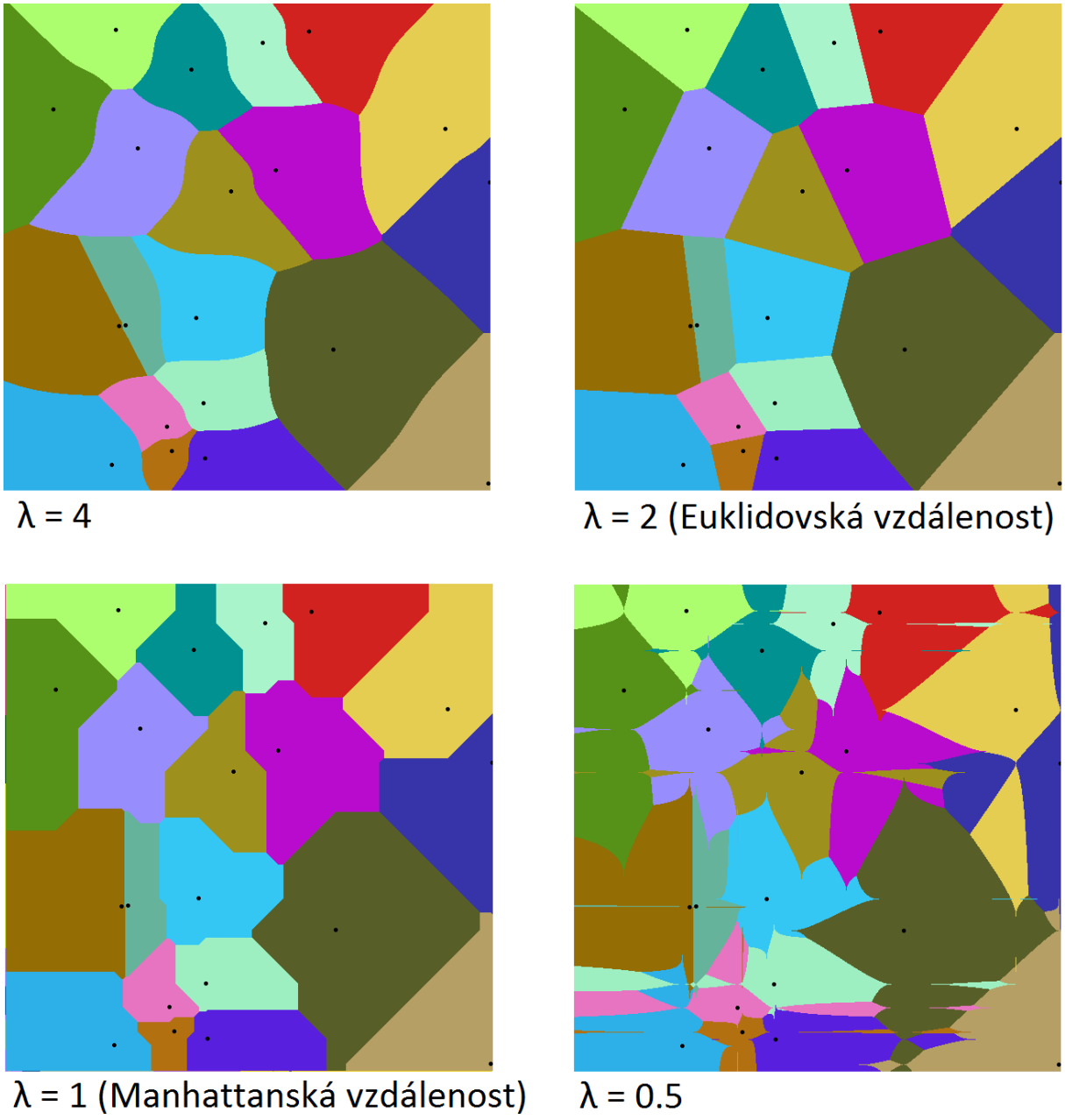
- Manhattanská vzdálenost - je inspirována pravoúhlou uliční sítí na Manhattanu. Jedná se o sumu horizontálních a vertikálních cest mezi dvěma body. Je vždy větší nebo rovna nule. Manhattanská vzdálenost je speciálním případem Minkowského vzdálenosti s parametrem $\lambda = 1$

  $$d_{ij} = \sum_{k=1}^{n} (x_{ik} - x_{jk})$$

- Minkowského vzdálenost - obecná metrika pro měření vzdálenosti.

  $$d_{ij} = \sqrt[\lambda]{\sum_{k=1}^{n} (x_{ik} - x_{jk})^\lambda}$$

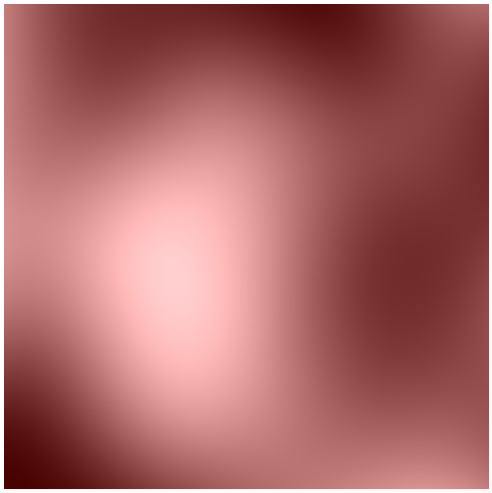
Na obrázku 11 vidíme Vorného diagram. Pro výpočet vzdálenosti mezi jednotlivými body je použita Minkowského metoda. Vidíme také různé nastavení parametru λ. Nyní si lze představit, že jednotlivé buňky představují různé biomy. Ze jedné buňky se může stát například poušť z další zimní krajina atd...



Obrázek 11: Různá hodnota parametru λ pro výpočet Minkowského vzdálenosti

Pro náš účel je důležité, aby tvary jednotlivých biomů nevypadaly stejně. Je taky potřeba, aby neměly ostré hrany, protože by pak ve světě byly vidět zlomy. Nejlepší efekt vytvoří různorodé a jemně zaoblené hrany. Dále je zásadní, aby spolu nesousedily biomy, které by spolu správně sousedit neměly. Jako například poušť se zimou. Tenhle problém se dá vyřešit tak, že svět pokryjeme teplotní mapou, vygenerovanou například pomocí Simplex noise. Mapu lze převést do grafické podoby, jak vidíme na obrázku 12. Podle ní pak umístíme jednotlivé biomy. Jiným řešením může být to, že předem určíme, jaké biomy mohou s sebou sousedit, a na základě těchto

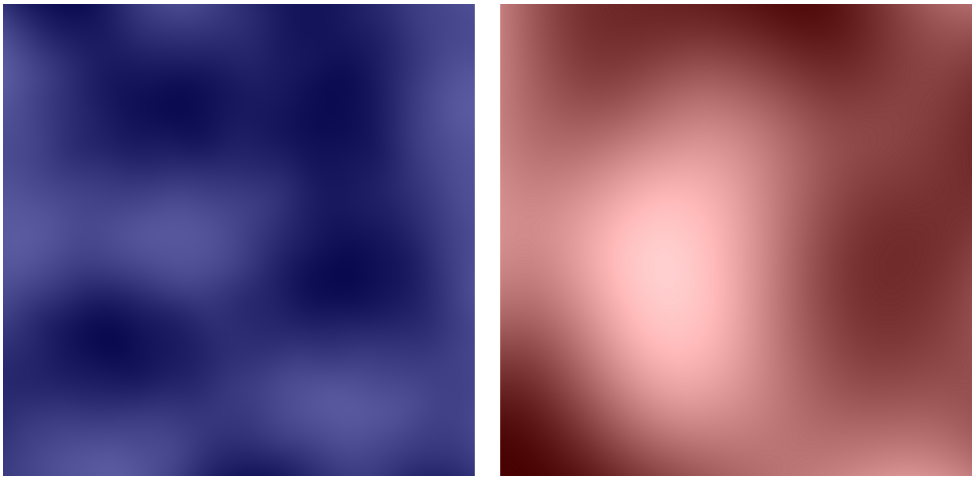
pravidel je rozdělíme.



Obrázek 12: Teplotní mapa (Bíla barva značí místa s nejvyšší teplotou a černá s nejnižší)

2.4 Vlhkost a teplota

Jiným možným způsobem, jak obarvit povrch, je řídit se podle určitých parametrů. Pokud se budeme inspirovat naší planetou, může to být například vlhkost a teplota. Pomocí Simplex noise můžeme vygenerovat teplotní a vlhkostní mapu. Grafickou podobu výsledných map, vidíme na obrázku 13.



Obrázek 13: Teplotní a vlhkostní mapa

Podle souřadnic z našeho světa si vytáhneme hodnoty vlhkosti a teploty z nově vytvořených map. Pomocí těchto hodnot následně vybereme barvu povrchu podle obrázku 14. V pravém horním rohu najdeme zelenou barvu, jelikož se jedná o nejvyšší teploty a nejvyšší vlhkosti. Naopak v levém dolním rohu ty nejnižší. Vpravo dole je žlutá barva pouště, jelikož se jedná o vysokou teplotu, ale nízkou vlhkost.

Obrázek 14: Barvy terénu podle teploty a vlhkosti [4]

2.5 Struktury

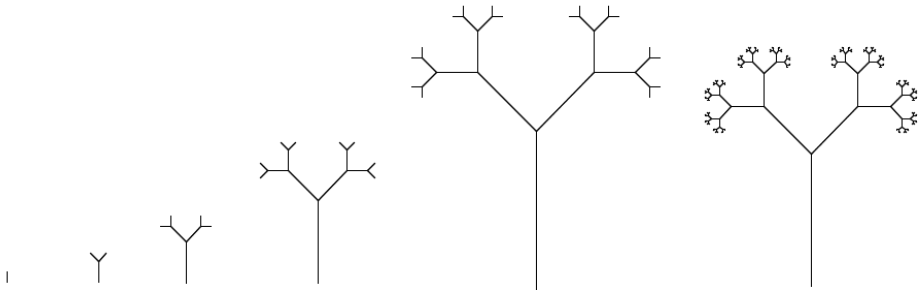
Jakmile máme vygenerovaný a obarvený terén, můžeme jej obohatit o různé struktury. Pokud se budeme opět inspirovat naší planetou, může se jednat například o stromy, jezera, řeky a budovy.

2.5.1 Stromy

Dnes již existuje spousta programů a nástrojů, pomocí kterých jsme schopni vytvořit reálně vypadající stromy a rostliny. Jedná se například o ATree3D [6], případně GrowFX [7] a spousty dalších.

Mezi nejstarší a snadno implementovatelné algoritmy pro generaci rostlin a stromů se řadí L-systém [5]. Jde o variantu klasické formální gramatiky, s tím rozdílem, že přepisovací pravidla jsou zde aplikována paralelně. Zápis je velmi podobný formální gramatice. Máme zde abecedu, konstanty a pravidla. Pomocí takové gramatiky a nějakého základního grafického provedení jsme schopni vytvořit jednoduché rostliny a stromy **??**lSystemTree). Výsledek se dá převést i do 3D podoby. Výsledný efekt je ale příliš závislý na daném pravidlu, a pokud tedy chceme mít mnoho

různých druhů stromů a rostlin, musíme také vytvořit mnoho různých pravidel, které je dále potřeba převést do grafické podoby.



Obrázek 15: Grafická reprezentace L-systému [5]

Jiným a jednodušším způsobem je definování základních typů stromů. Předem si vytvoříme základní schéma několika různých stromů, u kterých pak můžeme měnit základní vlastnosti jako například velikost koruny, výšku a šířku kmene atd... Zdá se to být jednoduchý postup, který nedokáže vytvořit reálně vypadající krajinu, ale podobně a hlavně dobře to tak funguje v Minecraftu. Hlavní výhodou je zde rychlost, není totiž potřeba žádným složitým algoritmem generovat tvar stromů, jelikož jejich základní kostry máme předem vytvořené.

Pro vytvoření krajiny, obsahující stromy, samotná generace nestačí, potřebujeme je taky rozumně po světě rozmístit. To můžeme udělat plně náhodně a rozházet je tak po celém světě. V takovém případě, ale najdeme například listnaté stromy v zimní krajině, kaktusy na louce atd...

Je tedy vhodné definovat, na jakém povrchu může jaký strom vyrůst. Například na trávě listnaté stromy, na písku kaktusy atd... Takovým způsobem, nejsme schopni vytvořit oblast, která bude představovat les s mnoha stromy a oproti tomu louku pouze s několika.

Pro tento případ je nejvhodnější použít rozdělení světa podle biomů 2.3. Pro každý biom můžeme definovat jaký strom a hlavně s jakou pravděpodobností se zde může vygenerovat. Můžeme tak vytvořit například biom „louka“, který obsahuje pouze travnaté bloky, na nichž se může s malou pravděpodobností vygenerovat listnatý strom a oproti tomu biom „les“, který obsahuje ty samé bloky, ale zde se mohou vygenerovat různé stromy s mnohem větší pravděpodobností.

2.5.2 Budovy

Na internetu lze nalézt spousty algoritmu pro generaci různých bludišť a katakomb. Bohužel téměř žádný pro generaci budov a následného rozdělení na místnosti a chodby. Z tohoto důvodu jsem se tedy rozhodl pokusit vytvořit vlastní algoritmus.

Základní myšlenkou algoritmu je možnost vygenerovat 2D schéma, ze kterého bude jasné, kde se nacházejí jednotlivé místnosti, chodby, dveře a okna. Vlastně takový zjednodušený půdorys. Důležité je, aby se schémata od sebe lišila. Schéma by tedy mělo být zcela náhodné a ve většině případů unikátní.

Nejdříve je potřeba definovat si základní konstanty:

- Maximální a minimální šířka a výška budovy
- Maximální velikost místnosti
- Velikost chodby, dveří, oken a zdí
- Výška budovy

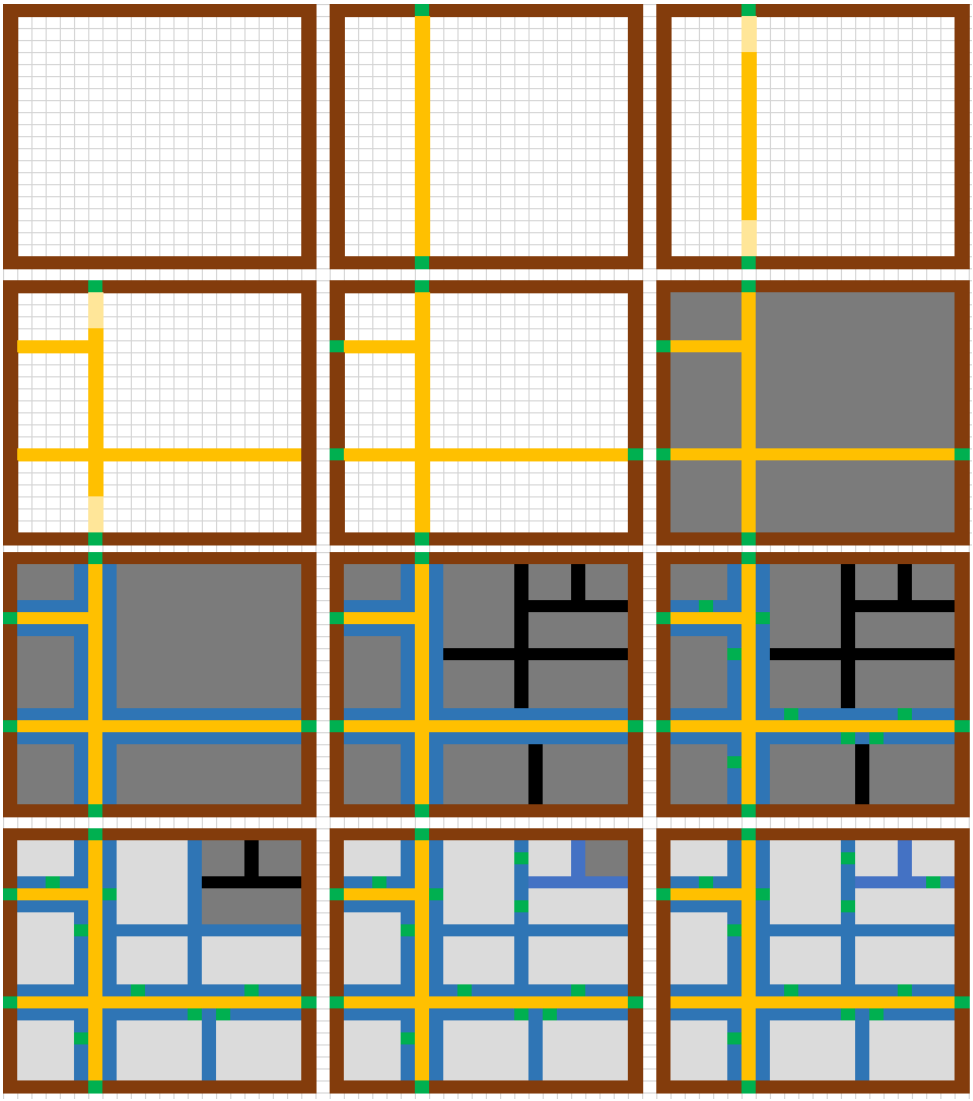
Na obrázku 16 vidíme grafický postup algoritmu. Ten se dá rozdělit do následujících kroků:

1. krok Určíme náhodně šířku a výšku půdorysu, samozřejmě v rozmezí námi zvolených konstant.

2. krok Z náhodného krajního místa protneme schéma základní chodbou. Náhodné místo zvolíme kousek dál od okraje. Na začátku a na konci chodby vytvoříme dveře. Na obrázku je vidíme jako zelený čtverec.
3. krok Ze základní chodby si vytýčíme oblast pro budoucí chodby. Tato oblast se nachází 15 % od začátku a konce chodby. Podle obrázku je to ta výraznější žlutá část.
4. krok V této oblasti pak náhodně vybereme náhodný počet míst, ze kterých povedou další chodby. Tyto chodby mají opačný směr a je zde 50 % pravděpodobnost, že chodba bude protínat základní chodbu skrz na oba kraje půdorysu.
5. krok Na konci, případně obou koncích, nově vzniklých chodeb vytvoříme další dveře.
6. krok Zbylá místa označíme jako místnosti. Podle obrázku tmavě šedé plochy.
7. krok Tyto plochy ohraničíme zdí, pouze tam kde sousedí s chodbou. Na obrázku je vidíme jako modré linky.

8. krok Podle konstanty, určující maximální velikost místnosti, rozdělíme zdí z náhodného krajního místa místnost na dvě místnosti. Takto pokračujeme, dokud každá místnost nebude splňovat maximální velikost. Tyto zdi si označíme jinou barvou, podle obrázku černou.

9. krok Jelikož je důležité, aby se z každé místnosti dalo dostat nějakou cestou ven z budovy, je potřeba místnosti propojit dveřmi, které povedou až k některé z chodeb. Toho docílíme tak, že pro každou místnost zkontrolujeme, zda obsahuje modré zdi. Pokud ano, na náhodném místě modré zdi vytvoříme dveře a černé zdi dané místnosti změníme na modré. To značí, že z místnosti vede cesta do chodby. Tento krok opakujeme, dokud nebudou všechny černé zdi obarveny na modro.



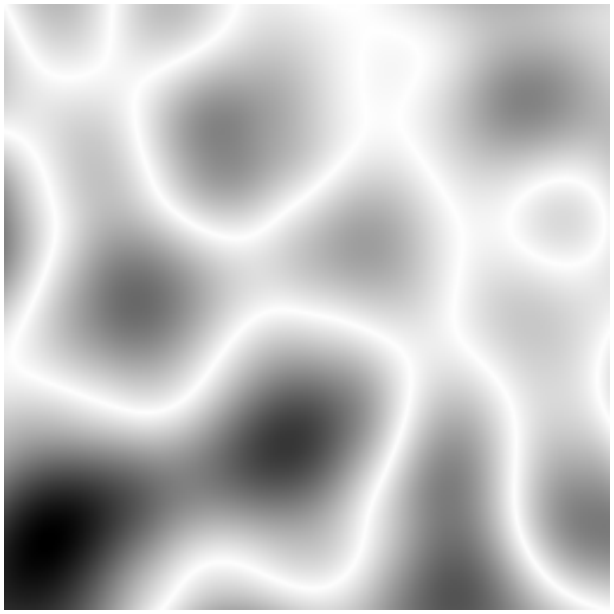
Obrázek 16: Grafická ukázka algoritmu pro generaci budov

2.5.3 Řeky

Řeky jsou nedílnou součástí světa, a proto by neměly chybět ani ve světě Kosmosu. Pro vygenerování řek existuje několik způsobů.

Jedním z nich je takový, kde vybereme náhodně dva body, kde první bod je vysoko umístěn a druhý zase nízko, nejlépe ve vodě. Poté tyto body spojíme přímkou, kterou pak zvlníme pomocí kolmých vektorů. Tento algoritmus, který jsem sám vymyslel a implementoval, je detailněji popsán v mé bakalářské práci [2].

Dalším ze způsobů může být využitím šumových funkcí. Pokud použijeme modifikovanou verzi Perlin noise, která se nazývá Ridged noise [8]. Při správném nastavení parametru této funkce jsme schopni vytvořit zaoblené křivky, představující řeky. Na obrázku 17 vidíme jejich tvar. Problém tohoto způsobu tkví v tom, že zde není dodržen přirozený princip řek. Řeky zde totiž proudí náhodným směrem, bez ohledu na výšku terénu. I když se tato metoda pro řeky příliš nehodí, je na druhou stranu vhodná pro cesty. Cesty se mohou ubírat náhodnými směry a nezáleží na tom, jestli se shora dolů nebo naopak.



Obrázek 17: Výsledek šumové funkce Ridged noise pro generaci řek

Pokud tedy chceme, aby řeky proudily správným směrem a to se shora dolů, můžeme tuto vlastnost využít při generaci. Ve světě najdeme náhodný vysoko umístěný bod. Poté vybereme jeden z okolních bodů, podle toho, který má nejnižší výšku. Tímto směrem se bude řeka ubírat a postupně tak vybíráme stále nižší a nižší body. Takto pokračujeme, dokud se nedostaneme k vodě, případně nějaké prohlubně, kde řeka může skončit. V případě, že výškový rozdíl mezi dvěma body řeky je příliš vysoký, je potřeba vytvořit vodopád.

2.6 Živé bytosti

V žádném světě by neměl chybět život. Ať už se jedná o lidi nebo zvířata, neměli by zde chybět. Pro jejich vytvoření máme hned několik možností.

Můžeme v některém z 3D grafický nástrojů, jako například Blender nebo 3DS Max, vytvořit 3D model podle svých představ. Poté jej rozpohybovat, tak že k němu přidáme animace.

Na internetu najdeme spousty 3D modelů s již vytvořenými animacemi, které jsou zdarma ke stažení. Zde to sice nebude přesně podle naších představ, ale ušetříme tím spoustu času.

Ve hře No Man's Sky 2.1.1 jsou tvorové generování procedurálně. Můžeme se tímto inspirovat a vytvořit několik různých částí a končetin, které následně spojíme do jednoho celku a vytvoříme tak unikátní tvory.

Poté co máme model vytvořený, musíme ho do světa vhodně umístit. To můžeme udělat podobně jako u stromů. Definujeme na jakém povrchu, případně biomu se může daný model vygenerovat. Dále ještě můžeme specifikovat jeho vlastnosti a chování. Například jeslí je agresivní a bude na hráče útočit, nebo se bude jen po světě procházet. Někteří mohou umět létat, jiní zase plavat atd...

3 Implementace procedurálního generování

Některé z výše popsaných metod a postupů jsem se pokusil využít pro tvorbu světa v projektu Jiný Kosmos. Tímto navazuju na svou semestrální práci, ve které bylo hlavním cílem vytvořit funkční podobu simulačního prostředí Jiný Kosmos. Šlo především o zprovoznění základního vykreslování bloků, podobně jako je tomu v Minecraftu. Dále pak o základní fyziku, pomocí které se dá ve světě pohybovat a kolidovat s okolními objekty. Na tomto projektu pracuje paralelně více studentů, kde má každý zadaný specifický úkol.

3.1 Terén

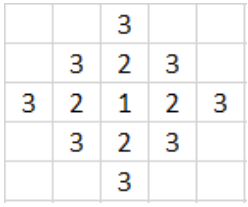
Pro generaci terénu jsem použil již výše zmíněný Simplex noise 2.2.1. Jeho implementaci nalezneme ve třídě „SimplexNoise“. Algoritmus jsem převzal z [3] a upravil tak, aby bylo možné z něj vytvořit instance. Nachází se zde metoda „seed“, pomocí, které můžeme nastavit náhodné semínko a tím tak zamícháme permutační pole, které obsahuje hodnoty od 0 do 255, každou dvakrát. Pomocí něj se v metodě „noise“ vypočítávají hodnoty šumu. Toto pole se v průběhu už nikdy měnit nebude, což nám zajistí, že pro stejné x,y,z nám vrátí stejné hodnoty.

Metodou „generate3D“ vygenerujeme pro daný bod(x, y, z) hodnotu od -1 do 1. S tou je dále pracováno ve třídě „Noise“, kde se hodnota vynásobí maximální výškou. Jestliže je výsledek větší y, znamená to, že jsme pod povrchem. V takovém případě zvolíme vhodný blok, který zde umístíme. V opačném případě necháme prostor prázdný (vložíme zde prázdný blok).

O všechny instance třídy „Noise“ se stará „NoiseManager“, kde najdeme jak šumové funkce pro výšku tak například i pro teplotu, vlhkost atd... Každé nové instanci je potřeba nastavit parametry šumové funkce.

Když tedy víme, jestli vložit blok či nikoliv, je potřeba určit, v jakou chvíli blok vložit. Svět Kosmosu je tvořen z mnoha bloků, které jsou seskupeny do tzv. Sloupků („Chunk“). Jeden sloupek má velikost 16 bloků a výšku 64 bloků. O tom, kolik sloupků se má kolem hráče vygenerovat, rozhoduje parametr *visibleRadius* nacházející ve „gameconfig.json“. V případě, že je tento parametr nastaven na hodnotu 3, vygenerují se sloupky s maximální vzdálenosti 3 sloupky od hráče. Určení vzdálenosti je zobrazeno na obrázku 18, kde číslo 1 je sloupek, na kterém hráč stojí. Je tedy potřeba vygenerovat 13 sloupků. Generace každého sloupku probíhá na novém vlákně. Jeden sloupek představuje jednu instanci třídy „Chunk“. Ve chvíli kdy je zavolán konstruktor této třídy, je spuštěna generace sloupku. Ten si pro každé své x,y,z nechá vygenerovat blok a to tak, že zavolá statickou metodu „generateBlock“ ve třídě „TerrainGenerator“, která dále spolupracuje s již výše zmíněnou třídou „Noise“.

Tímto je tedy zajištěna postupná generace světa. Je samozřejmě, že veškeré sloupky musíme udržovat v nějaké kolekci, kterou je potřeba promazávat, v případě, že sloupek už není ve viditelné oblasti.



Obrázek 18: Výpočet vzdálenosti sloupku

Poté co víme, kdy a jestli vůbec vložit blok, je potřeba rozhodnout jaký.

3.2 Volba bloku podle biomu

Popis postupu je popsán v kapitole 2.3. Nejdříve je potřeba vytvořit několik základních biomů a určit, které bloky mohou obsahovat. Na obrázku 19 vidíme 4 základní biomy a také bloky, které obsahují. Horizontální hodnoty značí teplotu a vertikální výšku. Za pomocí teploty se zvolí vhodný biom (Zimní krajina, pouštní krajina...) a poté na základě teploty a výšky se vybere daný blok.

Teplotu lze určit tak, že svět pokryjeme teplotní mapou za pomocí šumové funkce. Všechny tyto funkce najdeme ve třídě „NoiseManager“, která se stará například o teplotu, výšku, vlhkost atd...

Seznam všech biomů nalezneme ve výčtovém typu „BiomeType“ 1.

Vše je děláno co nejvíce dynamicky. Pokud tedy chceme přidat jakýkoliv nový biom, není nic jednoduššího, než do tohoto výčtového typu přidat další řádek a specifikovat parametry. Nový biom se pak bude objevovat ve světě.

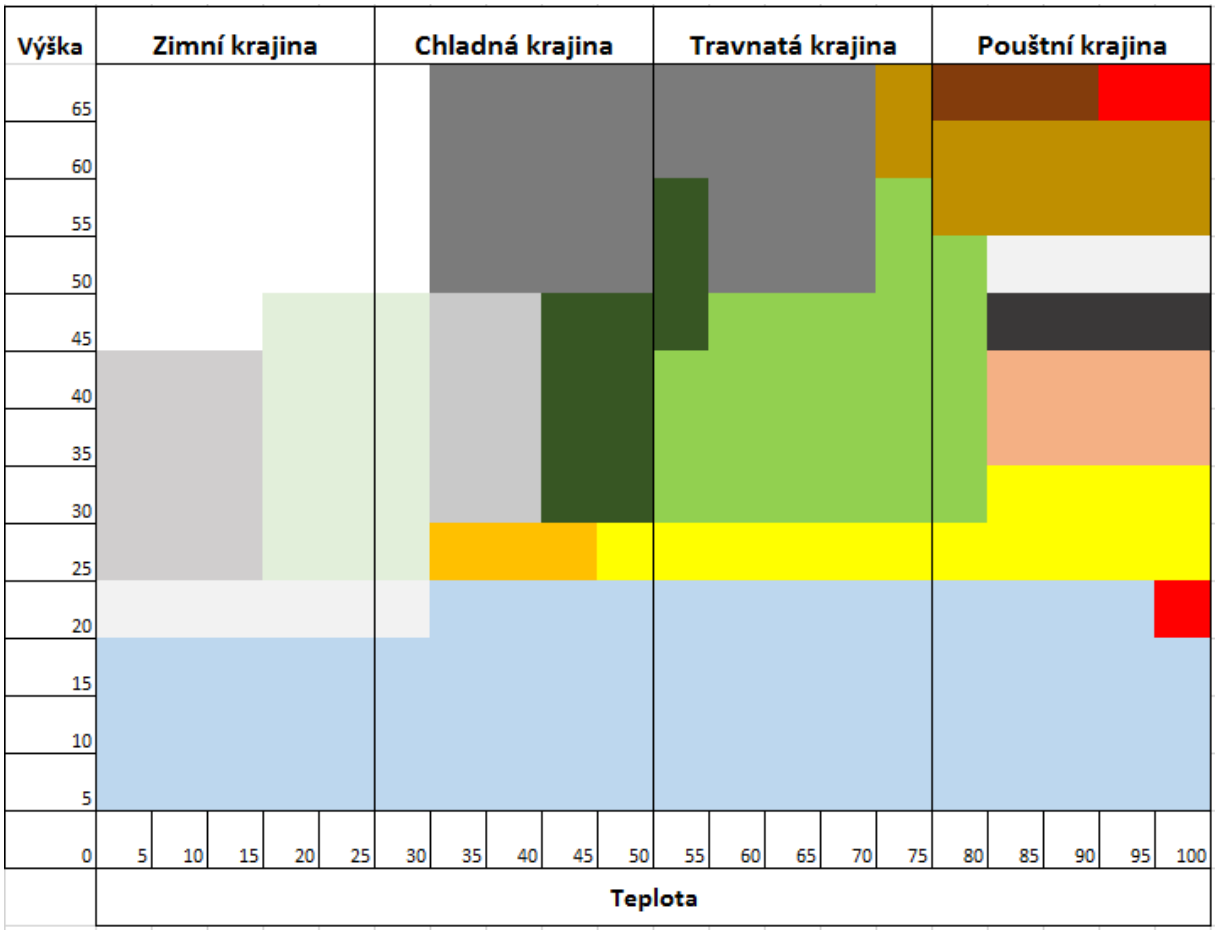
```
{
SNOW(0, 25, BlocksSet.SNOW_SET, 100, 3f),
COLD(25, 50, BlocksSet.COLD_SET, 100, 4f),
GRASS(50, 75, BlocksSet.GRASS_SET, 80, 25f),
FORREST(50, 75, BlocksSet.GRASS_SET, 120, 85f),
DESERT(75, 100, BlocksSet.DESERT_SET, 100, 10f);
}
```

Výpis 1: Seznam biomu z výčtového typu BiomeType

První dva parametry určují teplotu, pro kterou může být biom zvolen. Jak vidíme, tak biom „GRASS“ a „FORREST“ mají oba stejné teploty. Pro rozhodnutí, který z těchto dvou vybrat,

slouží čtvrtý parametr, který udává pravděpodobnost, s jakou se biomu vygeneruje. O volbu biomu, podle pravděpodnosti a teploty, se stará metoda „getBiomeByTemperatureAndChance“. Nejdříve se vyberou vhodné biomy. Například pro teplotu 30 by to byl GRASS a FORREST. Poté se sečtou pravděpodobnosti všech vhodných biomů. Pro tento případ tedy 80 + 120 = 200. Na závěr se pro výsledek 200 vygeneruje náhodné číslo od 0 do 200 pro výběr výsledného biomu. Poslední parametr udává úroveň zalesnění 3.4.

Třetí parametr je odkaz na seznam bloků. Tyto seznamy najdeme ve výčtovém typu „BlocksSet“. Jsou zde rozděleny bloky podle výšky a teploty přesně tak, jak vidíme na obrázku 19.



Obrázek 19: Biomy

Poté, co máme seznam všech biomů, můžeme je po světě vhodně rozdělit. Pro tento účel jsem využil Voroného diagram. Jeho implementaci nalezneme ve třídě „Voronoi“. Poté co je diagram vygenerován, každá jeho buňka obsahuje id biomu. Jeden biom je reprezentován instancí třídy „Biome“. V této třídě najdeme typ biomu a také metodu, která nám vrátí blok podle teploty a výšky ze seznamu bloků daného typu biomu. Všechny vytvořené biomy nalezneme ve třídě „BiomeGenerator“. Zde je udržována mapa, jejíž klíčem je id biomu a hodnotou samotná instance

Biomu. Když se tedy v „TerrainGenerator“ vybírá blok nejdříve se z „Voronoi“ vytáhne id biomu. Následně se toto id pošle do metody „getBiome“ v „BiomeGenerator“ kde se vytáhne instance „Biomu“ z mapy. V případě, že biom s takovým id neexistuje, vytvoří se nový a přidá se do mapy.

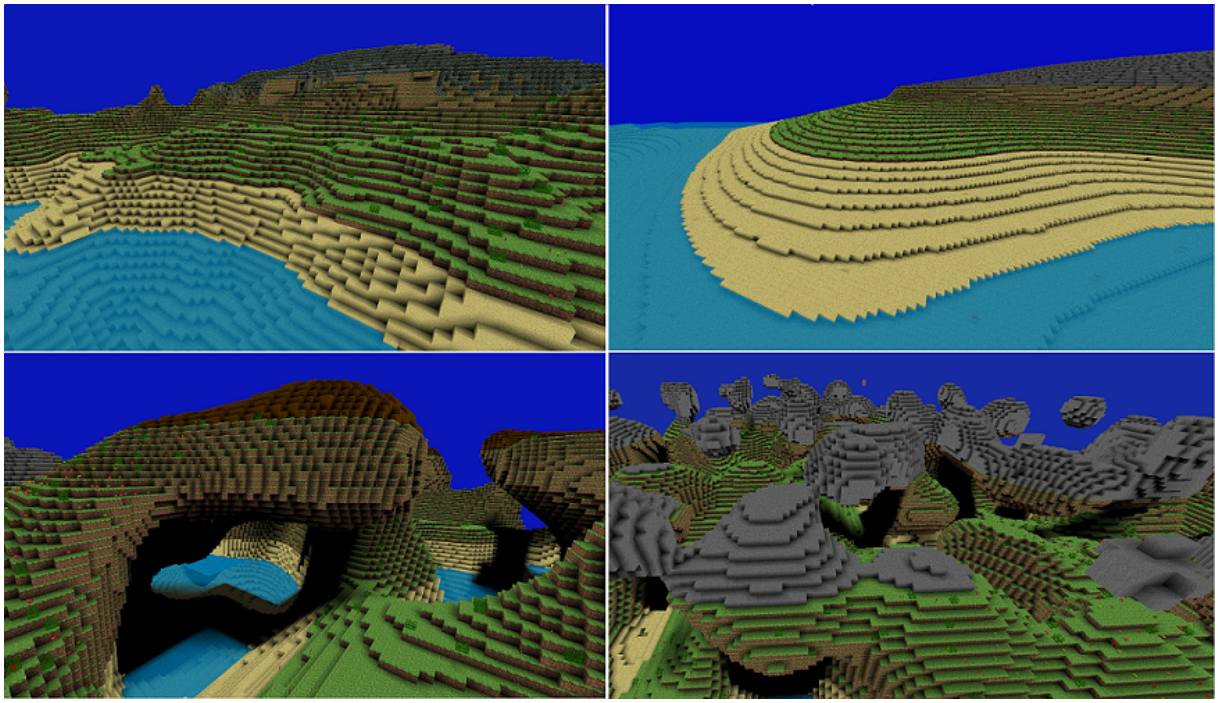
Jak si lze všimnout, konstruktor třídy „Biom“ obsahuje nejenom typ biomu, ale i typ terénu. V případě že pro „SimplexNoise“ nastavíme hodnoty frekvence, amplitudy a oktáv vždy stejné bude terén sice náhodný a rozmanitý, ale v podstatě typově stejný. Jestliže jednou zvolíme hodnoty tak, aby byl terén hodně členitý, pak takový bude vždy. Stejně tak když nastavíme hodnoty takové, aby byl hladký, tak takový bude stále. Proto jsem zde zavedl typy terénu. Pokud tedy máme svět rozdělený podle biomu, můžeme tak jednoduše pro každý biom určit jiný typ terénu.

Výpis všech typů terénu nalezneme ve třídě „TerrainType“ 2. Jednotlivé typy se liší pouze v jiném nastavení hodnot pro šumovou funkci. Navíc také každý typ obsahuje pravděpodobnost, s jakou se může vygenerovat. Najdeme zde například klasický typ terénu s největší pravděpodobností. Pak pláně a vysoká pohoří s nižší pravděpodobností. A na závěr také „létající ostrovy“. Při správném nastavení frekvence a amplitudy začnou ve světě vznikat kusy terénu oddělené od země, proto ten název. Tento poslední typ terénu má nejnižší pravděpodobnost, a proto na něj lze narazit jen zřídka.

Opět je toto tvořeno s co největším důrazem na dynamičnost. Pro nový typ terénu stačí tedy jen přidat nový řádek, specifikovat parametry a pravděpodobnost. Nový typ terénu pak nalezneme ve světě.

```
{
NORMAL(50, 0.3f, 0.005f, 0.3f, 3),
SMOOTH_MOUNTAINS(8, 0.003f, 0.01f, 0.3f, 3),
LOT_MOUNTAINS(8, 1, 0.02f, 3, 1),
PLAIN(8, 0.05f, 0.003f, 0.3f, 3),
FLOATING_ISLANDS(2, 0.01f, 0.035f, 0.3f, 3),
}
```

Výpis 2: Seznam typů terénu z výčtového typu TerrainType

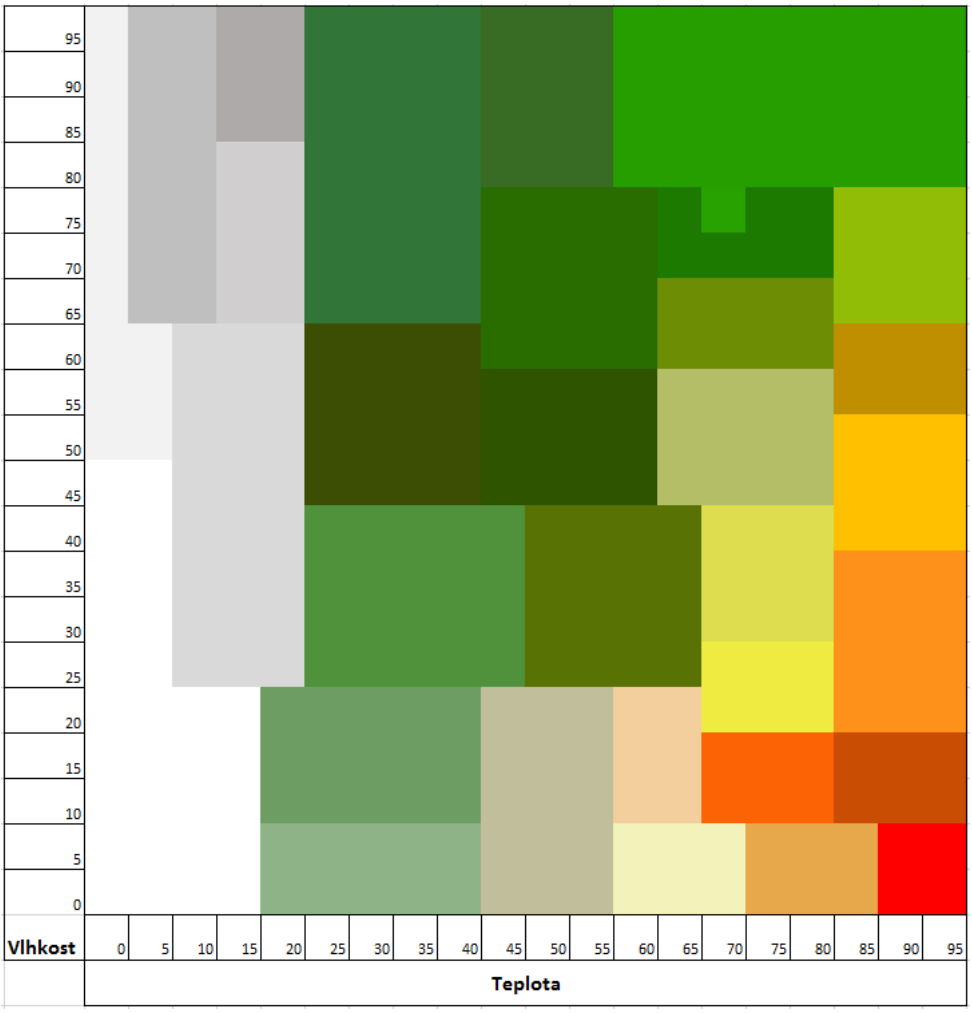


Obrázek 20: Různé typy terénu (Klasický, Pláň, Vysoká pohoří, „Létající ostrovy“)

3.3 Volba bloku podle vlhkosti a teploty

Jiným způsobem, jak vybrat blok, je například podle vlhkosti a teploty 2.4. Nejdříve je potřeba vytvořit teplotní a vlhkostní mapu. Parametry pro tyto šumové funkce najdeme v „NoiseParameters“. Instance těchto funkcí najdeme v „NoiseManger“, který se také stará o získání hodnot teploty a vlhkosti pomocí metod „getTemperature“ a „getHumidity“.

Ve chvíli, kdy máme alespoň dva parametry, podle kterých lze rozhodnout jaký blok vybrat, můžeme sestavit tabulku pro rozdělení bloků 21. Vertikální hodnoty značí vlhkost a horizontální teplotu. Podle těchto hodnot následně vybereme barvu bloku z tabulky. Například pro teplotu 90 a vlhkost 90 se dostaneme ke světle zeleným barvám, značící prales. Stejným způsobem tak obarvíme celou mapu.



Obrázek 21: Rozdělení bloků podle teploty a vlhkosti

3.4 Stromy

Jelikož i v reálném světě je každý strom jiný, dal jsem si i zde záležet, aby tomu tak bylo. Z tohoto důvodu nelze jejich tvar vygenerovat předem, a proto se musí generovat za běhu. Podle viditelné oblasti se propočítá, na který sloupek hráč dohlédne. Poté se o jeden sloupek dále vygenerují stromy. Tento algoritmus najdeme ve třídě „TreeGenerator“. O to, jestli stromy pro daný sloupek vygenerovat či nikoli, se stará metoda „tryGenerateTrees“, která je volána z „TerrainGenerator“ ještě před tím, než začne generovat terén. Nejdříve je potřeba zjistit, jestli náhodou už pro daný sloupek nejsou stromy vygenerovány. To se zjistí podle toho, jestli kolekce s názvem „generated“, obsahuje id sloupku, pro který chceme stromy vygenerovat.

Poté co víme, jestli stromy vygenerovat, je potřeba rozhodnout, kde, kolik a jaké. Jelikož víme, pro jaký sloupek strom generujeme, známe také přesné souřadnice této oblasti. Dále je pak náhodně vybrána pozice budoucího stromu, samozřejmě v oblasti daného sloupku.

V případě, že máme svět rozdělený podle biomů, jsme také schopni ovlivnit hustotu zalesnění. Jedná se již o výše zmíněný parametr „treeDensity“ v „BiomeType“. Podle aktuální pozice si vytáhneme biom, ve kterém se neházíme. Z biomu zjistíme typ biomu a z tohoto typu následně hustotu zalesnění. Čím větší je tato hodnota, tím více se vygeneruje stromů.

Na závěr je potřeba rozhodnout, o jaký typ stromů se bude jednat. Seznam všech stromů se nachází ve výčtovém typu „TreeType“ 3. Jeden typ obsahuje tři parametry. Prvním z nich je pravděpodobnost, podle které se vybírá druh stromu. Druhým parametrem je pole bloků, na kterých se může daný typ stromu vygenerovat. Posledním parametrem je pole biomů. Pouze v těchto biomech se pak bude daný typ stromu generovat. Zde je potřeba si dát pozor, aby se zvolené bloky nacházely také ve zvolených biomech, jinak by se nám tento typ stromu nikdy nevygeneroval.

O tom, který typ stromu vybrat, rozhoduje metoda „getTreeType“. Jejími parametry jsou typ bloku a typ biomu. Pro každý typ stromu se zjistí jestli obsahuje daný typ bloku a následně, zda obsahuje i typ biomu. Pokud jsou obě podmínky splněny, přidá se tento typ stromu do pole jako vhodný pro generaci. Následně se z tohoto pole vybere jediný typ podle jeho pravděpodobnosti.

```
{
CLASSIC_TREE(100, new byte[]{GrassBlock.ID, EarthBlock.ID, StoneGrassBlock.
    ID}, new BiomeType[]{BiomeType.GRASS, BiomeType.FORREST}),
TALL_TREE(80, new byte[]{RockySandBlock.ID, StoneGrassBlock.ID, StoneBlock.
    ID}, new BiomeType[]{BiomeType.COLD, BiomeType.FORREST}),
TALL_SNOW_TREE(80, new byte[]{SnowBlock.ID, IceGrassBlock.ID, SnowEarthBlock
    .ID}, new BiomeType[]{BiomeType.SNOW, BiomeType.COLD}),
DESERT_TREE(80, new byte[]{SandBlock.ID, DefaultSalmonBlock.ID,
    DefaultBrownBlock.ID}, new BiomeType[]{BiomeType.DESERT}),
CACTUS(100, new byte[]{SandBlock.ID, DefaultWhiteBlock.ID}, new BiomeType[]{
    BiomeType.DESERT});
}
```

Výpis 3: Seznam typů stromů z výčtového typu TreeType

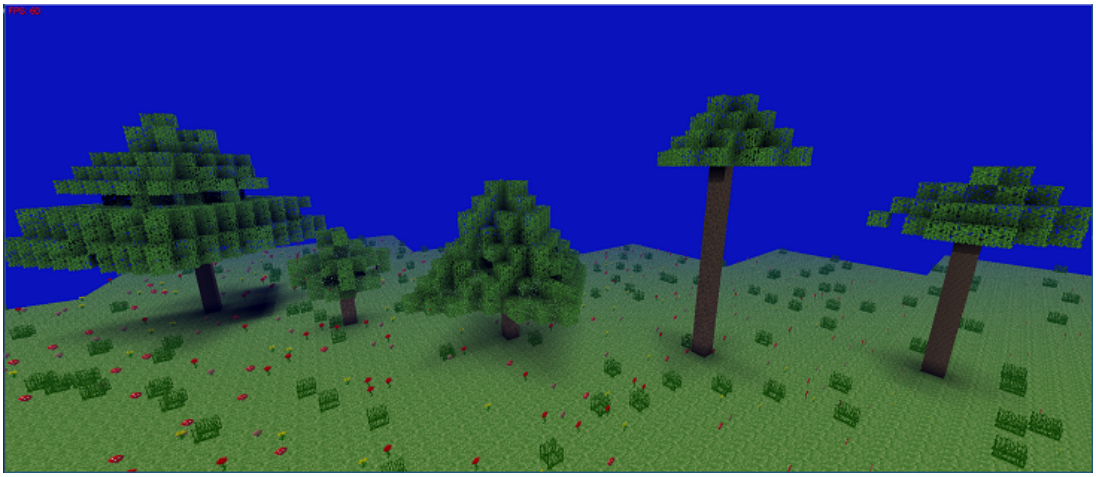
Když už víme kde a jaký typ stromu vygenerovat, je potřeba určit jeho tvar. To zajišťuje třída „TreeSchemeGenerator“. Každý typ stromu má zde vlastní algoritmus generující jeho vzhled. Najdeme zde základní veřejnou metodu „generateTree“, která podle typu stromu vygeneruje jeho tvar. Tato metoda vrací mapu, jejíž klíčem je souřadnice bloku a hodnotou je id bloku, jelikož jeden strom se může skládat z více různých druhů bloků.

I zde jsem kladl důraz na co nejvíce dynamický proces. V případě, že chceme přidat nový strom, můžeme přidat novou položku do tohoto výčtového typu. Dále je potřeba specifikovat

bloky a biomy, na kterých se bude nový strom generovat. A poté implementovat algoritmus generující tvar stromu. Nový strom se pak bude objevovat ve světě.

3.4.1 Klasické stromy

Pro generaci klasických stromů je zde metoda „generateClassicTree“. Nejdříve se vytvoří kmen stromu o náhodné výšce. Poté se určí náhodná výška koruny, která bude začínat v nejvyšším bodě kmenu. Pro symetrický tvar má výška a šířka koruny stejnou velikost. Koruna se postupně tvoří od spodního patra. Začíná se od vnitřního obvodu (první okruh 4 bloků) a postupuje se až k poslednímu největšímu, vnějšímu okruhu, jehož velikost se rovná šířce koruny. Kvůli tomu, aby každý strom vypadal jinak, se zde některé okruhy náhodně vynechají anebo se náhodně vynechá některý z bloků. Následně se šířka koruny zmenší o jeden a pokračuje se k dalšímu, vyššímu patru. Poté, co se dostaneme k poslednímu nejmenšímu patru, je algoritmus u konce a strom je hotový. Výsledek vidíme na obrázku 22.



Obrázek 22: Klasické stromy

3.4.2 Jehličnaté stromy

Jehličnaté stromy generuje metoda „generateConiferousTree“. Tento algoritmus funguje podobně jako generace klasických stromů 3.4.1. Rozdíl je zde v tom, že zde šířka koruny, není stejně velká jako výška koruny. Velikost patra se zde postupně nesnižuje, ale s určitou pravděpodobností se sníží, zvětší anebo zůstane stejná. Některá patra se nemusí vůbec vygenerovat.

Pro generaci zimních zasněžených stromů je použitý stejný algoritmus. Rozdíl je pouze v barvě bloků. Výsledek vidíme na obrázku 23.

Obrázek 23: Jehličnaté stromy v teplém a studeném podnebí

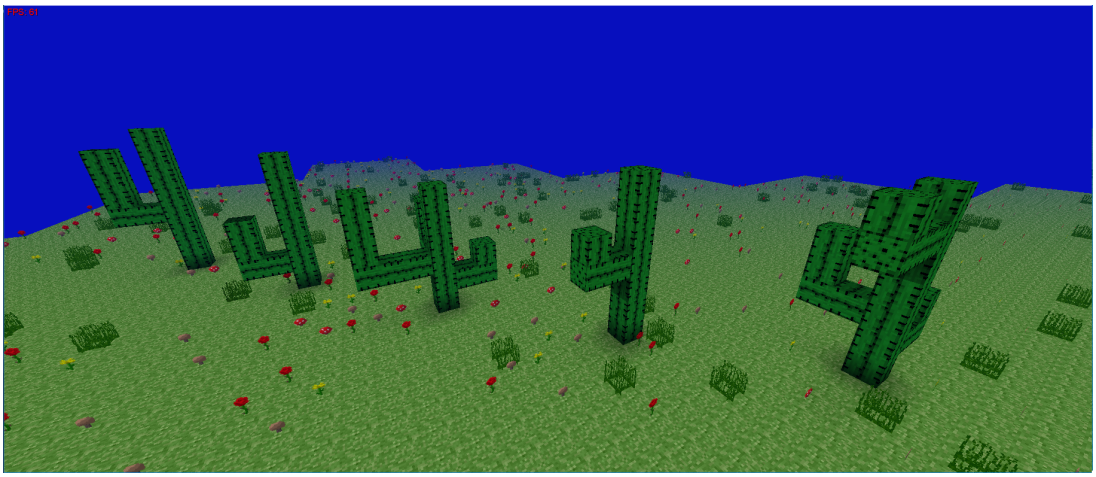
3.4.3 Pouštní stromy

Metoda „generateDesertTree“ se stará o generaci pouštních stromů. Tento strom je rozdílný v tom, že jeho koruna je mnohem nižší. Navíc se může stát, že se pro strom vytvoří druhá koruna začínající na nové větvi. Generace kmene a koruny probíhá podobně jako u klasických stromů 3.4.1. Akorát je zde mnohem širší a nižší koruna. S určitou pravděpodobností se rozhodne, jestli se bude tvořit druhá větev s další korunou. V případě že ano, zvolí se náhodné místo v kmeni, kde bude začínat nová větev. Poté se vybere náhodný směr nové větve. Na konci této nové větve se vytvoří nová koruna. Výsledek vidíme na obrázku 24.

Obrázek 24: Pouštní stromy

3.4.4 Kaktusy

Pro generaci kaktusu zde slouží metoda „generateCactus“. Jako první se vytvoří středový kmen. Dále se vygeneruje náhodný počet vedlejších větví. Pro každou z nich se určí náhodné místo, náhodný směr a náhodná velikost. Problém zde nastává ve chvíli, kdy se dvě větve vygenerují ve stejném směru a pod sebou nebo vedle sebe. Ve výsledku by se tyto větve pak spojily do sebe a vzhled by tak nevypadal příliš dobře. Proto je potřeba při přidávání každého nového bloku kontrolovat okolní bloky, jestli už náhodou něco není vygenerováno. Výsledek vidíme na obrázku 25.



Obrázek 25: Kaktusy

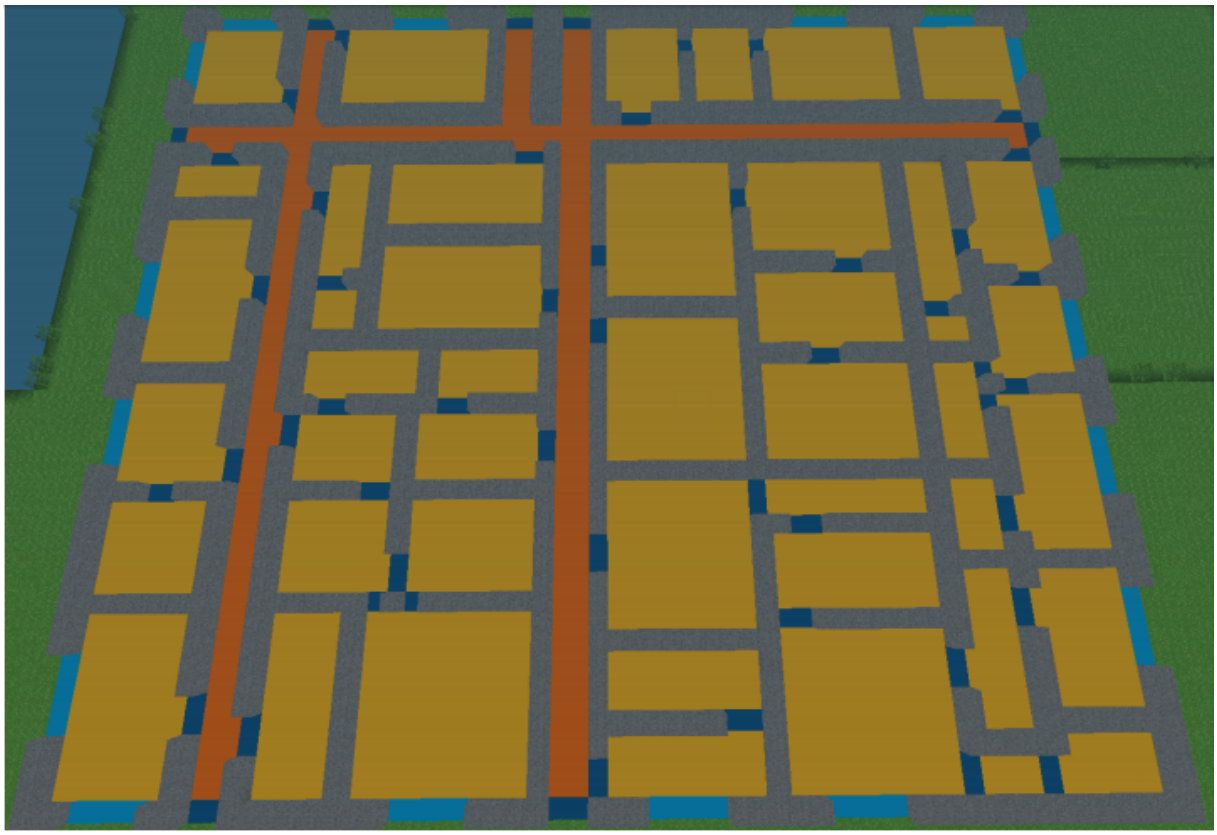
3.5 Budovy

Algoritmus pro generaci budov je krok po kroku popsán v kapitole 2.5.2. Jeho implementaci najdeme ve třídě „BuildingGenerator“. Vše začíná metodou „createBuilding“, pro kterou specifikujeme výšku a šířku. Tato metoda vrací dvourozměrné pole čísel, kde každé číslo udává, o jakou část budovy se jedná (dveře, místnost, chodba...). Nejdříve se začne s prázdným polem, které je následně rozděleno na chodby pomocí metody „createHall“. Zde se rozhodne kolik a jaké chodby bude výsledná budova mít. Některé chodby můžou protínat celou budovu, další můžou končit v chodbě jiné.

Jakmile jsou chodby hotové, zavolá se metoda „parseToRooms“. Zde se prohledá ono pole, a každý volný prostor, oddělený chodbou, se označí jako místnost. Pro každou místnost je vytvořena nová instance třídy „Room“. Tato třída obsahuje, jak schéma budovy, tak i informace o souřadnici místnosti a její velikosti. V případě, že místnost je příliš široká nebo vysoká, rozdělí se na menší místnosti (vytvoří se další instance). V případě, že jsou již všechny místnosti rozděleny do správných velikostí, je potřeba je propojit tak, aby se dalo projít do každé z nich.

Každá zeď sousedící s chodbou je označena jako průchozí, tedy vhodná možnost pro dveře. Všechny vnitřní zdi jsou označeny jako neprůchozí, nejsou tedy vhodné pro umístění dveří. Následně se pro každou místnost zkontroluje, zdali její okraj je tvořen průchozí zdí. V případě, že ano, vytvoří se v této zdi dveře na náhodném místě. Poté se všechny zdi této místnosti označí jako průchozí. Takto se pokračuje do doby, dokud nejsou všechny zdi a tedy i místnosti průchozí.

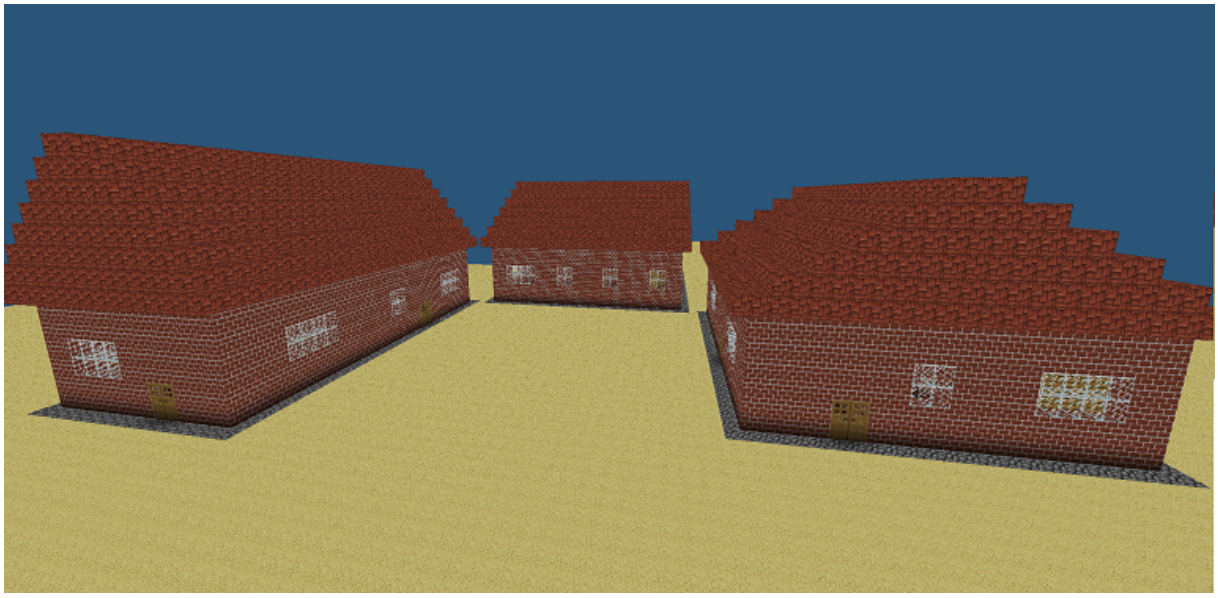
Místnosti obsahující vnější zdi budovy jsou vhodné pro vytvoření oken. V náhodných místnostech obsahující tyto zdi se vytvoří na náhodných místech okna. Ve chvíli kdy je celé dvourozměrné pole vyplněno, je hotový půdorys budovy. Grafickou ukázku tohoto půdorysu vidíme na obrázku 26.



Obrázek 26: Grafická reprezentace půdorysu budovy

Jelikož je každá budova jiná, nelze je vygenerovat předem a poté pouze rozmístit. Postupně při procházení světa se vytváří a ukládají pozice obdélníkových ploch pro budovy. Ovšem půdorys budovy je vygenerován až ve chvíli, kdy se generuje typ bloku v daném obdélníku. Jedno toto místo představuje jednu instancí třídy „BuildingPlace“, kde je také uloženo schéma půdorysu. Ve chvíli, kdy se generuje typ bloku, provede se dotaz, jestli blok neleží náhodou v tomto místě. V případě že ano, zavolá se metoda „getBuildingBlock“ s parametry x, y, z udávající polohu bloku. Podle polohy se vytáhne hodnota ze schématu budovy určující část budovy. V případě,

že se jedná o dveře a výška bloku je menší než 3, vrátí se blok typu dveře. Takto vzniknou dveře dva bloky vysoké. Podobným způsobem jsou pak tvořeny okna, zdi, střecha atd... Finální vzhled budovy vidíme na obrázku 27.



Obrázek 27: Budovy vygenerovány vlastním algoritmem

Jednotlivé místnosti budovy mohou být určitého typu. Například se může jednat o knihovnu, skladiště atd... Seznam všech těchto typů je umístěn ve výčtovém typu „RoomType“ 4. Každá místnost nemusí být vyplněna nábytkem, a proto zde najdeme i prázdný typ. Každý z nich má určitou pravděpodobnost, s jakou se může vygenerovat. Pro každou stranu zde najdeme typ nábytku, který může místnost obsahovat. Lze tak specifikovat, jaký nábytek bude mít přední strana a jaký zase zadní strana místnosti.

```
{
LIBRARY(10, FurnitureType.LIBRARY_F, FurnitureType.LIBRARY_R, FurnitureType.
    LIBRARY_B, FurnitureType.LIBRARY_L, null),
STOCK(10, FurnitureType.CHEST_F, FurnitureType.CHEST_R, FurnitureType.
    CHEST_B, FurnitureType.CHEST_L, null),
NONE(20, null, null, null, null, null);
}
```

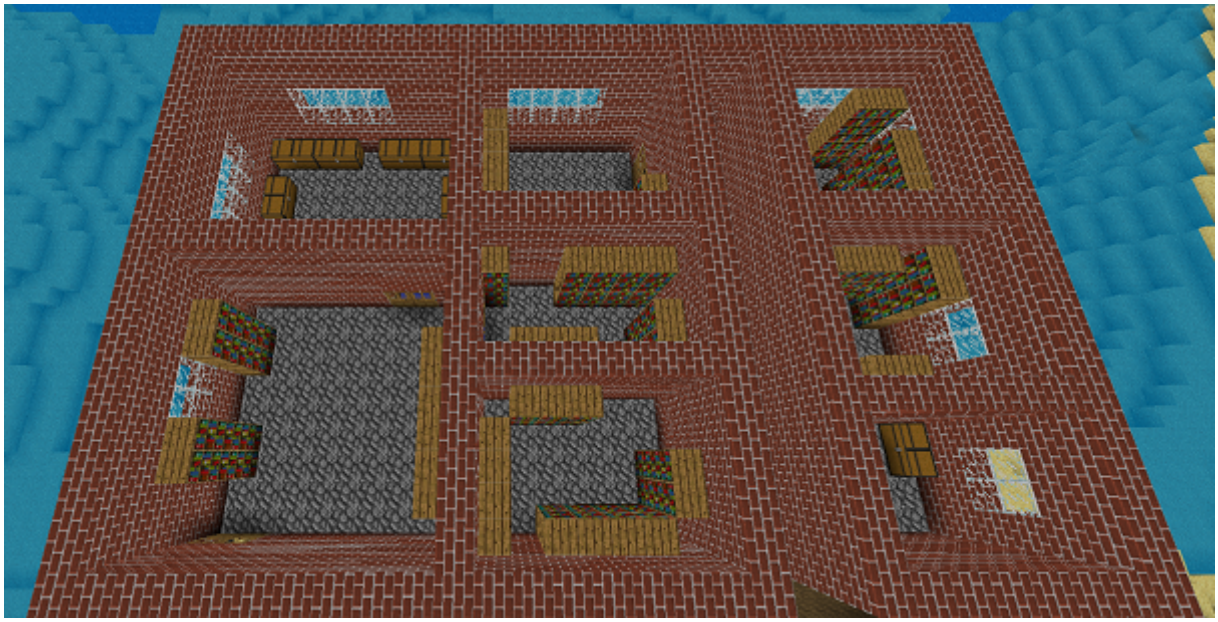
Výpis 4: Seznam typů místností z výčtového typu RoomType

Seznam všech typů nábytku najdeme ve výčtovém typu „FurnitureType“ 5. Obsahuje parametr hustota. Čím vyšší je toto číslo, tím více nábytku tohoto typu najdeme v místnosti. Dále je zde výška a pole bloků, z nichž je daný nábytek tvořen.

```
{
LIBRARY_L(40, 4, new BlockPos[]{new BlockPos(0, 0, 0, LibraryLeftBlock.ID),
    new BlockPos(0, 0, 1, LibraryLeftBlock.ID)}),
CHEST_F(20, 1, new BlockPos[]{new BlockPos(0, 0, 0, ChestRightFrontBlock.ID)
    , new BlockPos(1, 0, 0, ChestLeftFrontBlock.ID)});
}
```

Výpis 5: Seznam typů nábytku z výčtového typu FurnitureType

Aplikace jednotlivých typů místností a následné umístění nábytku probíhá ve třídě „Room“, konkrétně v metodě „applyRoomType“. Zde se vybere náhodný typ budovy, ze kterého je následně vytažen seznam nábytku. Postupně je pak každý díl nábytku umístěn podle pozice. Pro ukázku jsou zde pouze dva typy: knihovna a skladiště. Výsledné místnosti vidíme na obrázku 28.



Obrázek 28: Různé typy místností vybavené nábytkem

3.6 Řeky

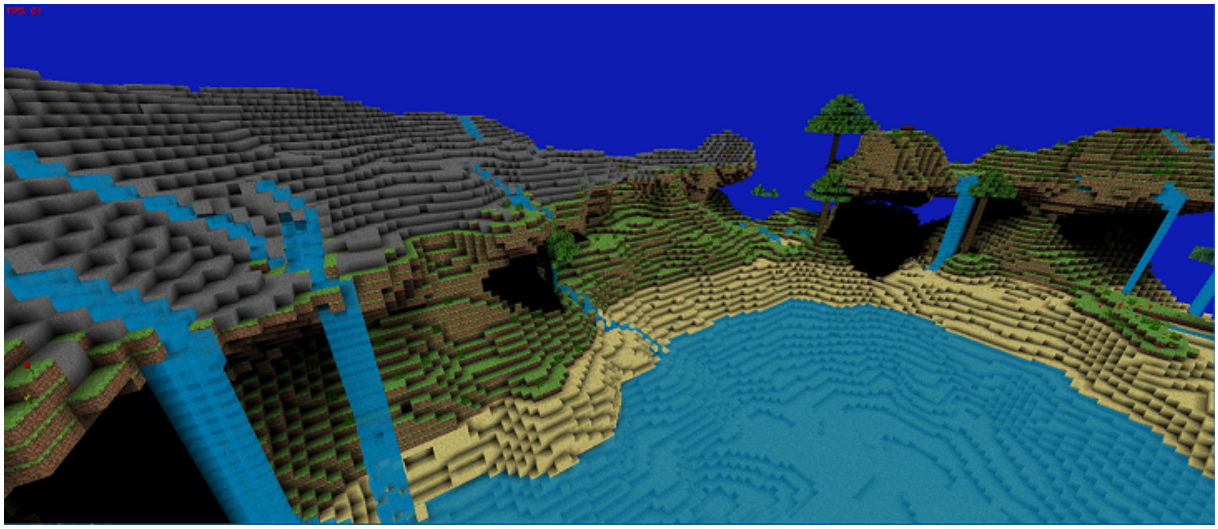
V kapitole 2.5.3 je popsán algoritmus pro generaci řek. Jeho implementaci nalezneme ve třídě „RiverGenerator“. Řeky se generují postupně podle toho, jak hráč prochází svět. V metodě „generateRivers“ se vybere několik náhodných bodů, které se uloží do pole a seřadí podle výšky. Následně se určí náhodný počet řek pro danou oblast. Podle tohoto počtu se vybere několik nejvyšších bodů, jež budou představovat počátek řeky.

Samotná generace jednotlivých řeky se pak nachází v metodě „generateRiver“. Zde se od startovacího bodu, uloží hodnota výšky všech okolních bodů. Nedíváme se zde na přímou hodnotu výšky bloku, ale na hodnotu, jež vrací šumová funkce terénu. Výška bloku je totiž celé kladné číslo, kdežto hodnota šumové funkce je reálné číslo, určující výšku přesněji. Pro příklad můžeme mít dva bloky, jejichž výška je stejná, ale hodnota šumové funkce se může o pár desetinných míst lišit. Díky tomu lze určit výškový rozdíl i mezi stejně vysokými bloky a můžeme se tak snáz rozhodnout, kterým směrem se řeka bude ubírat. Jako další blok vybereme ten s nejnižší výškovou hodnotou a celý proces zopakujeme, dokud se nedostaneme k vodě.

V případě, že výškový rozdíl mezi aktuálním blokem a příštím je příliš vysoký, je potřeba vytvořit vodopád. O to se stará metoda „createWaterFall“. Jednoduše se zde vypočítá výškový rozdíl mezi těmito bloky, jež bude určovat výšku vodopádu.

Je potřeba také myslet na to, že řeka může být z jedné strany odkrytá. Nebudou zde žádné bloky, které by je ohraničily. Takže během generace je zapotřebí kontrolovat tyto okraje a popřípadě doplnit jednotlivé bloky.

Výsledné řeky a jejich vodopády vidíme na obrázku 29.



Obrázek 29: Ukázka řek a vodopádů

3.7 3D modely

Jeden 3D model představuje jednu instanci třídy „Creature“. Ta udržuje informace o pozici, rotaci, velikosti atd... Mimo to taky obsahuje instanci třídy „AnimationController“, který zde zajišťuje fungování animací. V neposlední řadě obsahuje i instanci „btRigidBody“, což je třída z rozšíření Bullet 4.1.5.

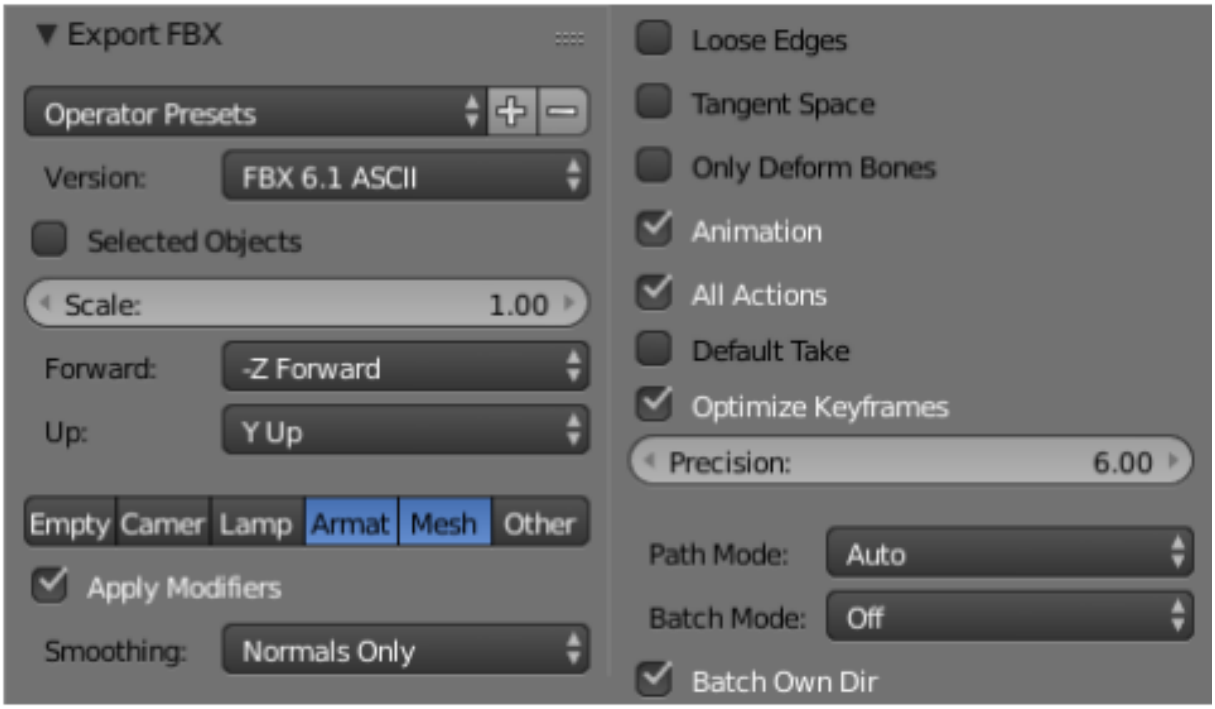
O generaci veškerých tvorů se stará "CreatureGenerator". Jelikož se tyto modely mohou pohybovat a animovat, musíme je ve smyčce „render“ neustále aktualizovat. Z tohoto důvodu je zde kolekce obsahující všechny instance třídy „Creature“. Každá položka této kolekce je pak s každým zavoláním „render“ aktualizována a vykreslena.

Seznam všech modelů, které ve světě najdeme, je umístěn ve výčtovém typu „CreatureType“ 6. Jsou zde vypsány vlastnosti jednotlivých tvorů, jako například to, na kterých typech bloků se mohou vygenerovat, případně ve kterých biomech. Dále jestli je onen tvor agresivní či nikoli.

```
{
KNIGHT(20, true, new byte[]{GrassBlock.ID, EarthBlock.ID, StoneGrassBlock.ID
    }, new BiomeType[]{BiomeType.GRASS, BiomeType.FORREST, BiomeType.COLD}),
ARCHER(10, false, new byte[]{GrassBlock.ID, EarthBlock.ID, StoneGrassBlock.
    ID}, new BiomeType[]{BiomeType.GRASS, BiomeType.FORREST, BiomeType.COLD})
    ;
}
```

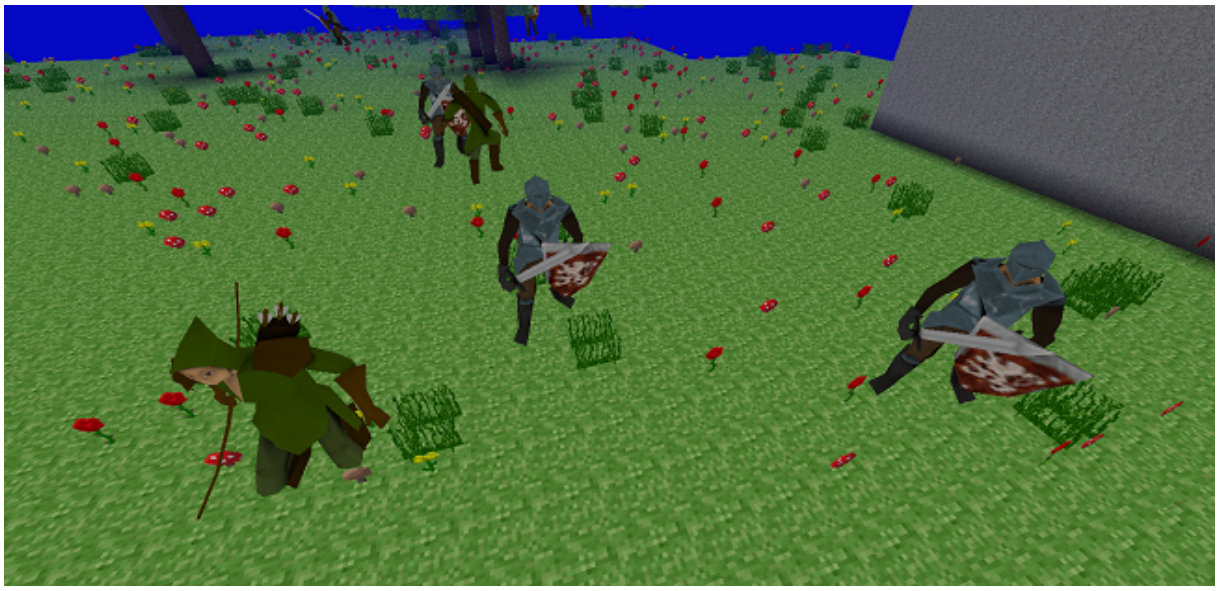
Výpis 6: Seznam tvorů z výčtového typu CreatureType

I zde je kladen důraz na dynamiku. Pro přidání nového modelu stačí jen do tohoto výčtového typu přidat nový řádek a specifikovat parametry. Samozřejmě je potřeba 3D model vložit do složky „models“ i s jeho texturami. Zde je menší problém s formátem výsledného modelu. Jako vzorový příklad jsem použil model rytíře a lučištníka dostupný z [12]. Nejdříve je potřeba model exportovat do formátu „.fbx“. Pro export z Blenderu [13] je potřeba použít specifické nastavení, jak vidíme na obrázku 30. Dalším krokem je „Fbx converter“ [14], který model převede do formátu „.g3db“. Výsledek je mnohem menší a přijatelnější pro plynulost hry. Pro převod stačí přes příkazovou řádku spustit „fbx.exe“ s parametry „-f, vstupní soubor, výstupní soubor“ (například takto: „fbx.exe -f knight.fbx knight.g3db“). Jakmile tohle vše splníme, výsledný model se bude objevovat ve světě.



Obrázek 30: Správné nastavení exportu z programu Blender do formátu „.fbx“

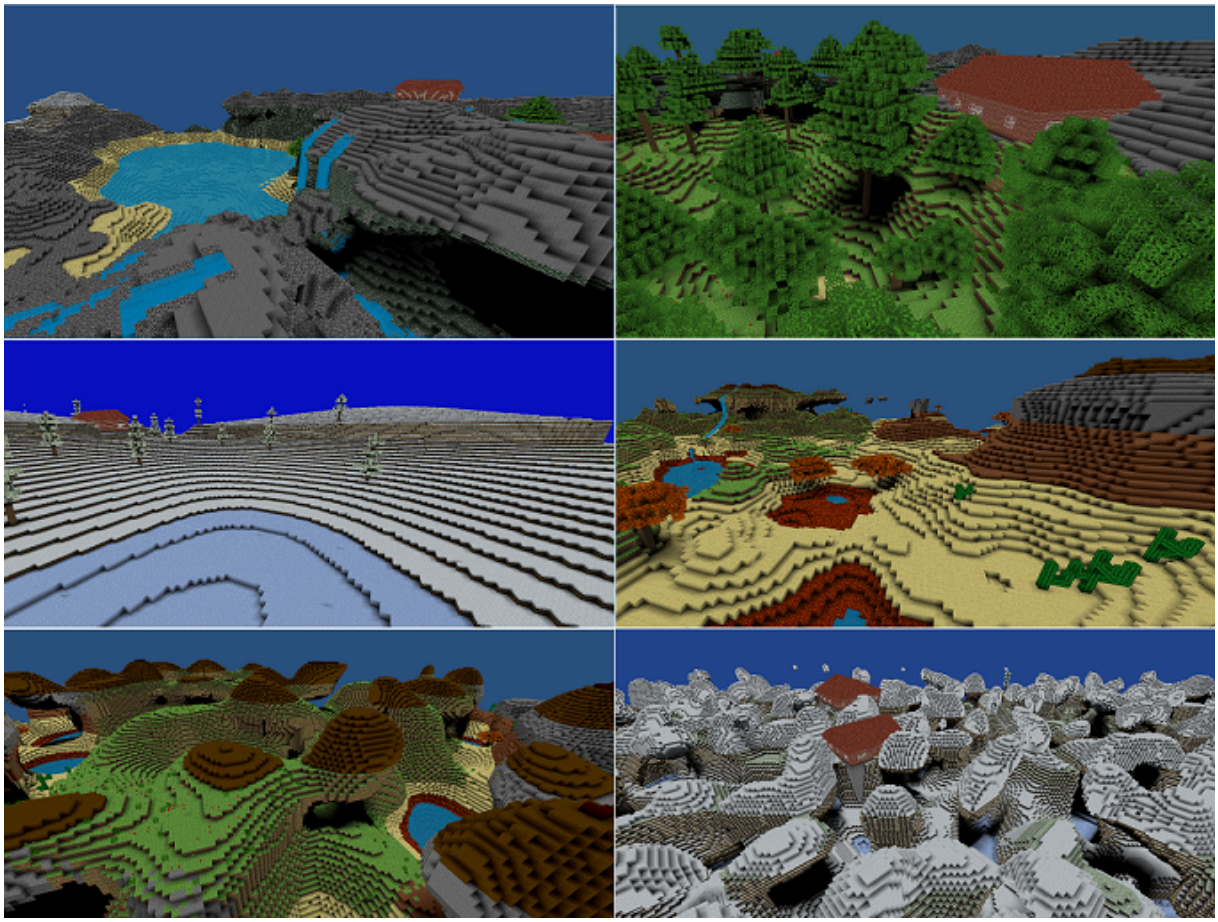
Po přídání nového prvku do výčtového typu se výsledný model bude generovat ve světě na blocích a v biomech, které jsme pro něj zvolili. Tvorové se pak budou náhodně pohybovat po světě. V případě, že jsme ho nastavili agresivním, na nás zaútočí, jakmile se k němu dostatečně přiblížíme. Ukázku tvorů vidíme na obrázku 31.



Obrázek 31: Ukázka vygenerovaných tvorů

3.8 Výsledek

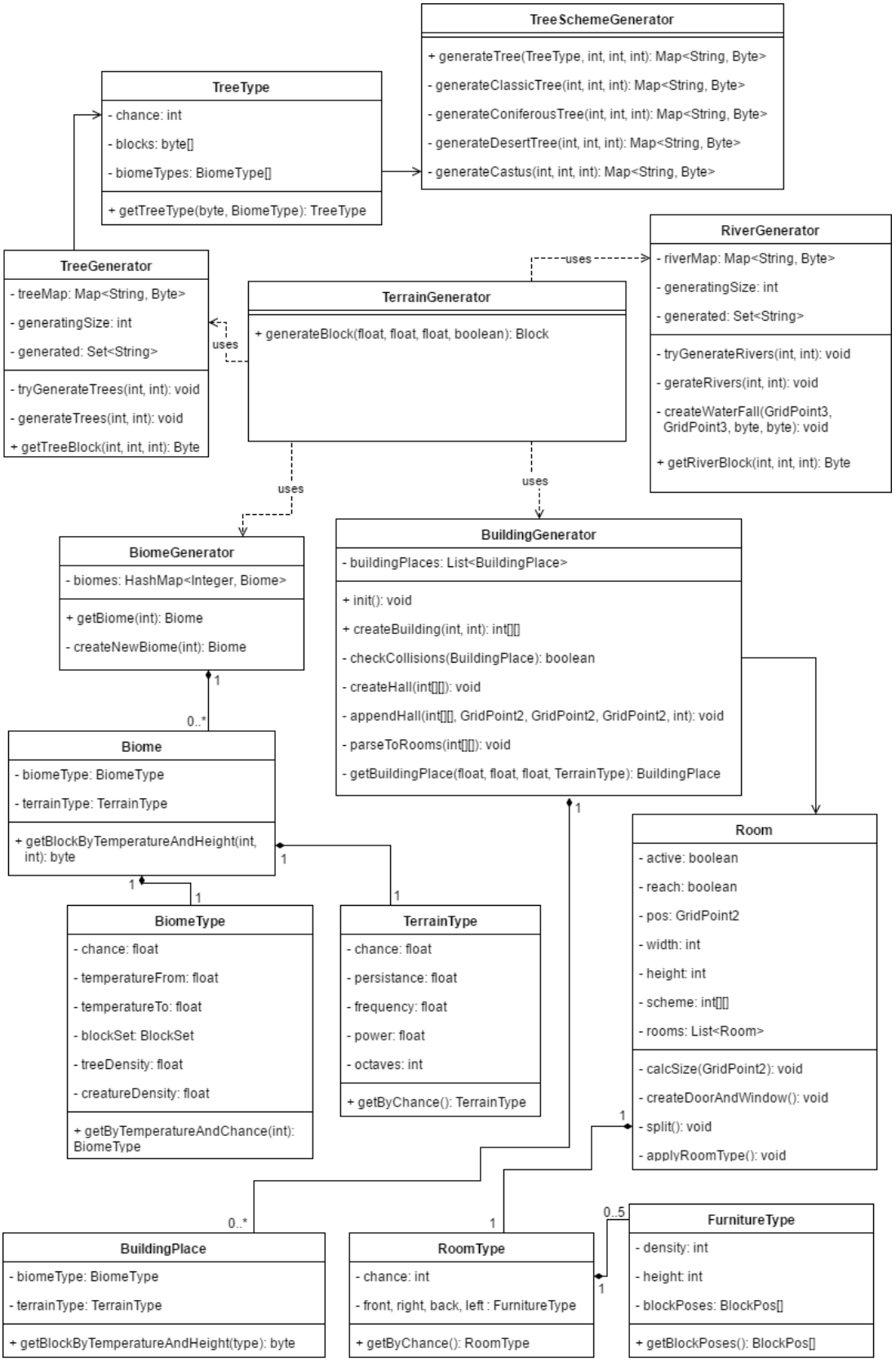
Implementací všech těchto postupů jsme schopni vytvořit 3D reálnou krajinu. Ano, vždy se dají najít nějaké nedostatky. Ty se ale postupným laděním a úpravami dají vypilovat do lepší podoby. Výsledek toho všeho vidíme na obrázku 32.



Obrázek 32: Výsledek procedurálního generování světa

3.9 Třídní diagram

Pro lepší přehled vidíme na dalším obrázku 33 třídní diagram již výše zmíněných generátorů. Jak lze vidět veškeré dění se děje ve třídě „TerrainGenerator“, která následně využívá ostatní generátory pro generaci stromů, řek, biomů a budov.



Obrázek 33: Zjednodušený třídní diagram generátorů

4 Implementace aplikace

I přestože se tato práce zaobírá především procedurálním generování, je i tak potřeba alespoň letmo popsat jakým způsobem aplikace funguje a jaké nástroje pro její vytvoření byly použity.

4.1 Použité technologie

4.1.1 LibGDX

LibGDX [10] [9] je rámec určený pro vývoj her. Je založený na OpenGL a je open source. Je stále vyvíjen a zlepšován rozsáhlou komunitou vývojářů. Jeho přední výhodou je, že je multiplatformní.

Podporované platformy:

- Windows
- Android
- Mac OS X
- Linux
- iOS
- HTML5

Ve výsledku tedy máme pouze jediný kód, který je spustitelný na všech těchto platformách. Pro každou z platforem nabízí podporu pro:

- Zvuk
  - Přehrávání hudby a zvuků (WAV, MP3, OGG)
- Vstup uživatele
  - Myš a klávesnice
  - Dotyk obrazovky
  - Detekce gest
- Matematiku a fyziku
  - Pomocné třídy pro vektory, matice a kvaterniony
  - Třídy pro geometrické tvary a detekci jejich kolizí
- Soubory
  - Čtení, zápis, kopírování a mazaní

- Výpis
- Ověření zda soubor nebo adresář existuje
- „Preferences“ funkce pro uložení nějakého nastavení

Dále nabízí podporu pro OpenGL ve třech úrovních vyspělosti:

- Nízká úroveň - Textury, odstíny, jednoduché vykreslování tvarů jako obdélníku, kruhu atd...
- Vysoká úroveň 2D API - Ortografická kamera, třída „SpriteBatch“ pro vykreslování obrázku a písma.
- Vysoká úroveň 3D API - Perspektivní kamera pro 3D projekci.

Podporuje i knihovny třetích stran jako například:

- Box2D - knihovna pro podporu 2D fyziky
- Bullet - knihovna pro podporu 3D fyziky
- Spine - 2D animace pomocí systému kostí
- Kryonet - podpora TCP a UDP komunikace
- FreeType - určený pro generaci fontů

4.1.2 Struktura projektu

LibGdx vytvoří strukturu projektu podle zvolených platforem. Jeden modul pro zdrojové soubory „core“ a pak jednotlivé moduly pro každou platformu, v našem případě jen pro desktop. V modulu „desktop“ se nachází pouze jedna třída „DesktopLauncher“ určená pro spuštění aplikace. Zde se nastaví verze OpenGL, šířka a výška aplikace atd... Zdrojová data jako například textury, zvuky, fonty atd... najdeme v baličku „assets“.

4.1.3 Gradle

Pro zajištění závislostí a sestavení projektu používá LibGDX nástroj Gradle. Jedná se vlastně o automatizační nástroj jako například Maven nebo Ant. Vývojář se tedy nemusí starat o knihovny, vše zajistí a stáhne Gradle. V projektu najdeme Gradle soubory s definicí závislosti. Jeden hlavní v kořenové složce a druhý v modulu „core“. V hlavním „build.gradle“ najdeme veškeré závislosti, čísla verzí a pravidla pro sestavení jednotlivých modulů. Jelikož v tomto projektu se používá rozšíření Bullet, je tato závislost přidána i zde 7.

```
project(":core") {
apply plugin: "java"

dependencies {
compile "com.badlogicgames.gdx:gdx:$gdxVersion"
compile "com.badlogicgames.gdx:gdx-bullet:$gdxVersion"
compile group: 'com.google.guava', name: 'guava', version: '15.0'
}
}
```

Výpis 7: Závislosti pro nástroj Gradle

4.1.4 IntelliJ IDEA

Projekt jsem vytvářel ve vývojovém prostředí Intellij IDEA. V původním zadání semestrálního projektu bylo sice napsáno, že máme používat Eclpise, ale tento projekt je snadno spustitelný v obou prostředích. Po otevření projektu ve vývojovém prostředí IDEA by měl Gralde vše stáhnout a nastavit automaticky. V případě, že by projekt nešel spustit kvůli neexistujícím souborům, je potřeba nastavit konfiguraci pracovního prostředí („Working directory“) na složku „\core\assets“. Pro spuštění v Eclipse je potřeba stáhnout do tohoto prostředí doplněk ”Gradle (STS) Integration for Eclipse“ z „MarketPlace“.

Toto vývojové prostředí jsem si vybral, jelikož mám s ním již dlouholeté zkušenosti a jednoduše jsem si na něj zvykl.

4.1.5 Bullet

Rozšíření Bullet je pomocná knihovna třetí strany určena pro výpočet fyziky ve 3D. Stará se především o detekci kolizí objektů, jak pohyblivých, tak statických. Dále zajišťuje gravitaci.

Její využití najdeme především ve třídě „Physics“. Důležitý je zde proměnná „collisionWorld“, do kterého se přidávají veškeré objekty světa. V metodě update, která běží v nekonečné smyčce, se zavolá metoda „stepSimulation“ na „collisioWorld“, kde se propočítají veškeré kolize.

4.1.6 Texture Packer

Pro udržení většího množství textur po hromadě je použit nástroj „gdx-texturepacker“ [11]. Tento nástroj zabalí všechny textury do jedné a vytvoří k stejnojmenný soubor, kde jsou uloženy pozice a velikost jednotlivých textur.

4.2 Základní třídy

Vše začíná ve třídě „DesktopLauncher“ kde se zavolá konstruktor třídy „LwjglApplication“ a jako parametr se zde vloží instance třídy „MainClass“. To je třída, kterou LibGDX vytvoří při vytváření projektu. Odtud je už zbytek práce na nás.

„MainClass“ dědí ze třídy „Game“, což je třída využívána pro vývoj her a nabízí podporu změny obrazovek metodou „setScreen“. Zde můžeme jako argument poslat jakoukoliv instanci implementující rozhraní Screen, čímž zajistíme změnu obrazovky na jinou. Ve chvíli kdy nastane nějaká událost, GDX automaticky zavolá příslušné metody. Jako například:

- create - Volána při spuštění aplikace
- resize - Volána hned po metodě „create“, při změně obrazovky a při změně velikosti
- puase - Volána ve chvíli kdy uživatel opustí aplikaci a to tak, že jí například minimalizuje
- resume - Volána ve chvíli kdy se uživatel vrátí do aplikace
- render - Jedna z nejdůležitějších metod. V případě, že aplikace běží, je tato metoda volána nepřetržitě, v nejlepším případě 60krát za sekundu.

Po spuštěni aplikace, je automaticky zavolána metoda „create“ ve třídě „MainClass“. Zde se inicializují základní proměnné a jako hlavní obrazovka je nastavena instance třídy „SplashScreen“. Tato obrazovka obsahuje pouze název projektu. Během toho co je zobrazena se načítají textury, nastaví se fyzika, inicializují generátory, vytvoří kamera, vytvoří „Voxel“ a jako nová obrazovka se nastaví instance třídy „GameScreen“.

4.3 Voxel

Jádro celé aplikace. Ve třídě „GameScreen“ najdeme jeho instanci a také metodu „render“, ve které je nepřetržitě provolávaná metoda „voxel.render“. Zde se s každým provoláním aktualizuje pozice kamery a všech modelů. Dále se aktualizuje „ChunkManager“. Na konec se vše vykreslí pomocí „ModelBatch“, který je určený právě pro 3D objekty.

4.4 ChunkManager

Udržuje všechny vytvořené sloupky v mapě. Zajišťuje prioritu aktualizace sloupků, podle toho, který je potřeba co nejdříve vykreslit. Obsahuje velice užitečné metody jako „getBlockAtWorl-

dPosition“, která vrátí ID bloku nacházejícího se na dané pozici, stejně tak i metoda „get-ChunAtWorlPosition“, která vrací daný sloupek.

V případě potřeby se zde vytvářejí nebo odstraňují Sloupky. Slouží také pro načítání a ukládání sloupků.

4.5 Sloupek („Chunk“)

Udržuje informace o každém bloku zařazeném do onoho sloupku. Bloky jsou zde reprezentovány celým číslem, které odkazuje na typ bloku. Zde je postupně vytvářen každý blok po bloku v metodě „calculateChunk“. Po vytvoření všech bloků se provedou nezbytné kroky, jako kalkulace světla, aktualizace atd...

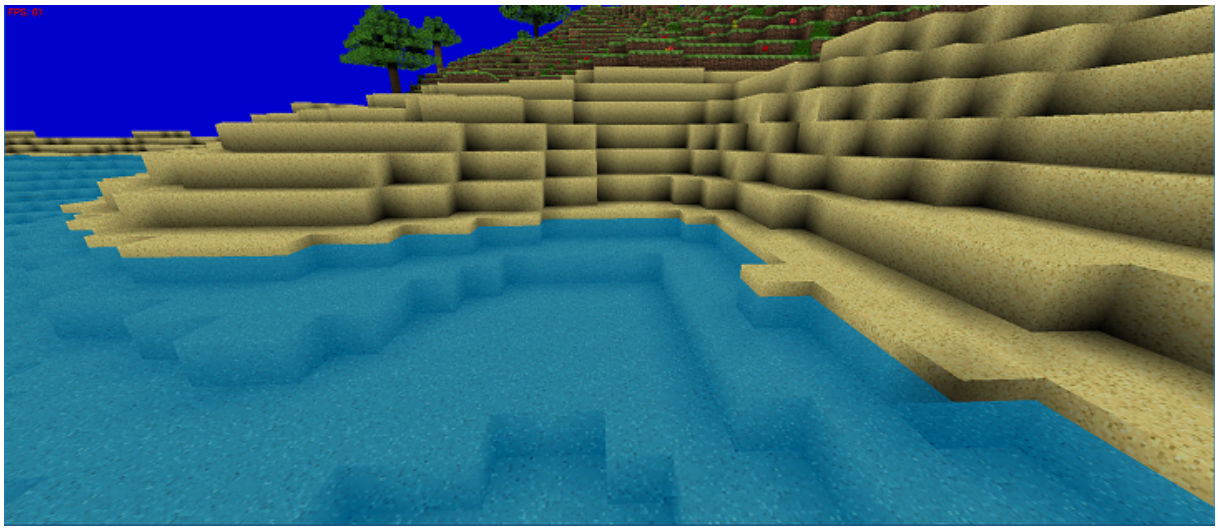
4.6 Block

Nejmenší prvek ve světě Kosmosu. Obsahuje informace o typu bloku, jeho texturách, jestli je kolizní s hráčem, průsvitnost atd... Počet instancí bloků je přesně takový, jaký je počet typů bloků (Balíček „block.types“). Ke každé instanci bloku je přiřazený vlastní „BlockRender“.

4.7 Vykreslování bloků („BlockRender“)

Stará se o vykreslení potřebných stěn bloku a správné vykreslení osvětlení bloku. U většiny bloků totiž není nutné vykreslovat všechny strany. Například není potřeba vykreslovat stěnu mezi dvěma sousedícími bloky, jelikož tato stěna nejde vidět. Je potřeba jí vykreslit až ve chvíli, kdy je jeden z bloků odstraněn.

Má několik rozšíření jako například „WaterRender“, který je určený pro vodu. Vykresluje jen horní stranu bloku a to jen v případě, že nad daným blokem vody se nenachází žádný jiný blok. Tuto horní stranu navíc sníží o pár pixelů, aby voda nebyla přesně zarovnaná s břehem 34. Další variací je pak „FlowerRender“ a „StrawRender“, které jsou určeny pro vykreslování rostlin a hřibů. V případě „FlowerRender“ je vykreslena pouze jedna strana, která je následně umístěna na střed spodní části, díky čemuž můžeme vykreslit rostliny, případně hřiby a další 2D objekty 35. „StrawRender“ pak vykresluje čtyři vertikální strany a každou z nich posune blíž ke středu. Díky tomu je možné vykreslit trávu 35.



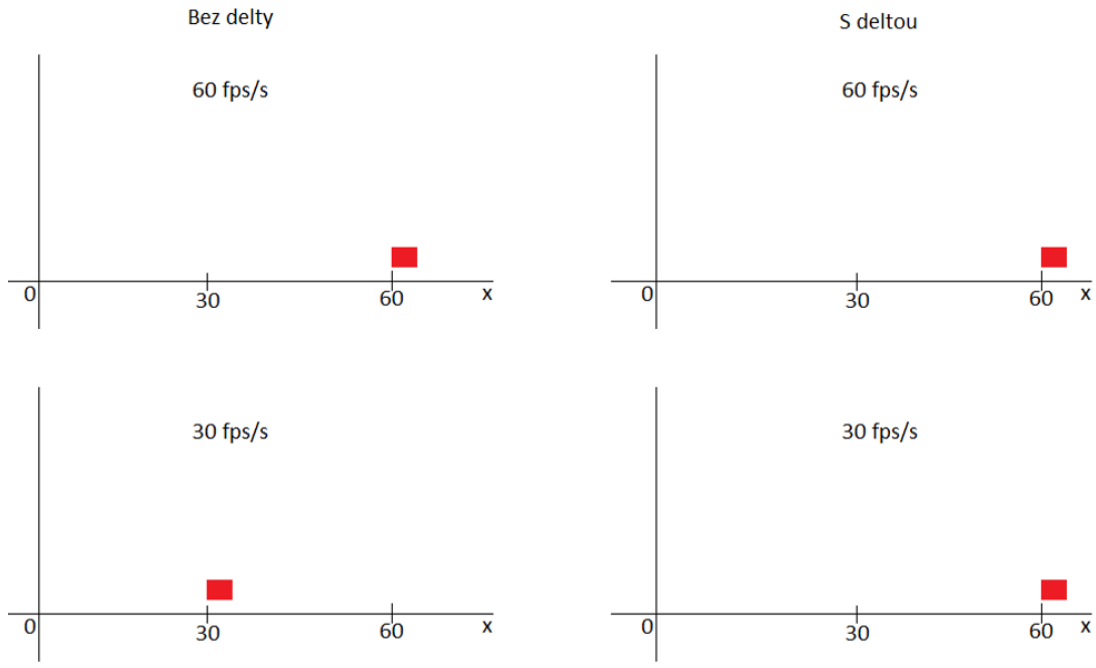
Obrázek 34: Voda vykreslena pomocí „WaterRender“

Obrázek 35: Květiny, hřiby a tráva pomocí „FlowerRender“ a „StrawRender“

4.8 Delta

Jak si lze všimnout, metody „render“, ať už v „GameScreen“ nebo „Voxel“, obsahují parametr „delta“. Jedná se o krátký časový úsek, který uběhne mezi dvěma zavoláním funkce „render“. Je to vlastně čas, který uběhl od posledního vykresleného snímku.

Delta má normalizační schopnost. Jako příklad 36 si lze představit, že chceme červený čtvereček posunout o 60 pixelů za sekundu. Vzhledem k tomu, že tato metoda je volána většinou 60krát za sekundu, stačí zde tedy čtvereček posunout o jeden pixel. Díky toho docílíme plynulého pohybu a náš objekt je na námi požadované pozici, přesně jak vidíme na obrázku vlevo nahoře. V případě, že máme pomalejší hardware a naše aplikace běží pouze 30 snímků za sekundu, se metoda render zavolá pouze 30krát za sekundu. Náš červený čtvereček se pak posune jen o 30 pixelů, jak taky lze vidět na obrázku vlevo dole. Tento problém nám právě pomůže vyřešit delta hodnota, která je při jednom snímku za sekundu rovná jedné. V nejlepším případě 60 snímku za sekundu by byla delta rovna 1/60. Pokud tedy chceme docílit toho, aby se nám čtverec vždy posunul o stejný počet pixelů, stačí tento posun vynásobit deltou $60x delta$. Při 60 snímcích by to tedy bylo: $60x(1/60) = 1$, tedy posun o jeden pixel s tím, že metoda se zavolá 60krát. Ve výsledku tedy 60 pixelů, jak můžeme taky vidět na obrázku vpravo nahoře. Při 30 snímcích by to bylo: $60 * 1/30 = 2$, tedy posun o dva pixely s tím, že metoda se zavolá 30krát. Ve výsledku tedy 60 pixelů, jak vidíme na obrázku vpravo dole. Za pomocí delty máme čtvereček vždy na správném místě, ať už by aplikace běžela v jakémkoliv počtu snímku za sekundu.



Obrázek 36: Ukázka využití hodnoty delta

5 Uživatelská dokumentace

Po spuštění aplikace se uživateli zobrazí načítací obrazovka s názvem projektu. Jakmile je vše načtené, hra se spustí a uživatel je vhozen do světa Kosmosu. V tomto světě se může pohybovat pomocí kláves „W, A, S, D“ a po stisku mezerníku také skákat. Pro ukázku je zde i možnost létání a to stisknutím kombinací kláves „ctrl + P“. Uživatel má možnost, přidávat bloky pravým tlačítkem myši. Typ vkládaného bloku, lze změnit posunutím kolečka myši. Veškěré bloky pak lze smazat levým tlačítkem.

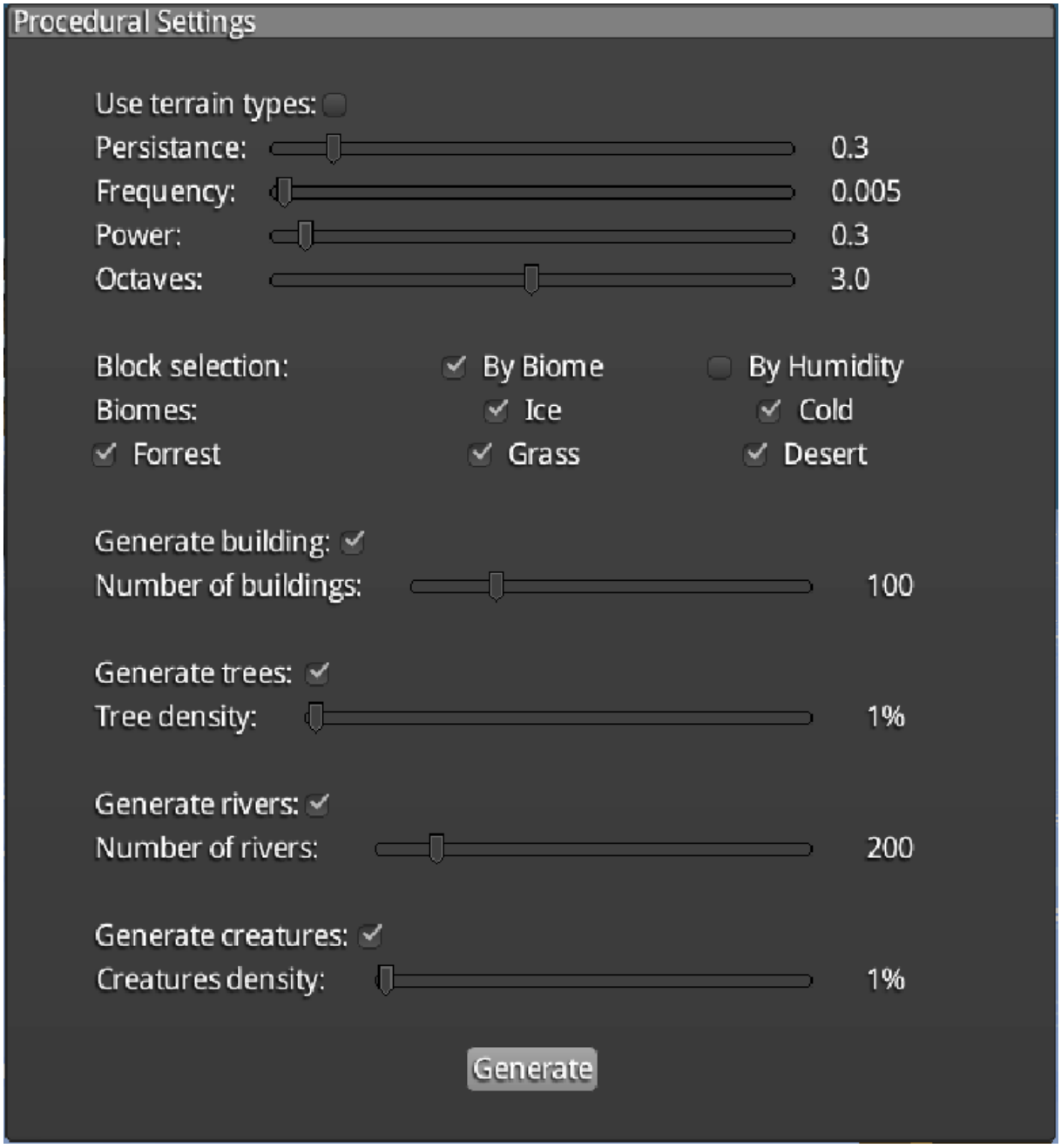
Najdeme zde i více nastavení jako například inventář atd... Tyto funkcionality jsem nedělal já, ale někdo jiný z vývojářů projektu Jiný Kosmos, a proto je zde nebudu popisovat.

5.1 Nastavení procedurální generace

Po stisknutí klávesy „G“ se vyvolá nabídka nastavení procedurální generace světa, jak lze vidět na obrázku 37. V případě volby „Use terrain types“ se bude typ terénu vybírat z již výše zmíněných a předem vytvořených nastavení 2. Nebo si případně může uživatel sám nastavit hodnoty parametrů pro šumovou funkci „Simplex Noise“. S frekvencí a persistencí doporučuji zacházet opatrně, jelikož výrazně ovlivňují tvar terénu. Stejně tak i s počtem oktáv, které při vyšším počtu výrazně zvyšují výpočetní složitost.

Dále máme na výběr způsob výběru bloků pomocí nastavení „Block selection“. Volbou „By Biome“ se budou bloky vybírat pomocí metody biomů, přesně jak je popsáno v kapitole 3.2. Volba „By Humidity“ je zde pouze pro ukázku jiného způsobu výběru bloků podle kapitoly 3.3. Zaškrtnutím této možnosti, pak nelze ve světě generovat budovy, stromy, řeky ani tvory. Ve světě Kosmosu je pět základních biomů. Pomocí nastavení „Biomes“ lze vybrat, které biomy se budou generovat a které ne.

Další nastavení určují v jaké míře a zdali vůbec se budou generovat budovy, stromy, řeky a tvorové. Po stisku tlačítka „Generate“ se veškeré nastavení promítne do světa, který je následně znovu vytvořen s již novými parametry.



Obrázek 37: Nastavení procedurální generace po stisku klávesy „G“

6 Závěr

V této práci jsem popsal a implementoval postupy, pomocí kterých lze vytvořit různorodý 3D terén. Za zmínku stojí generátor budov, který jsem sám vymyslel a implementoval. Jedná se o poměrně komplexní proces, který umí vytvořit půdorys budovy, včetně dveří a oken. Vše propojené tak, že se z kterékoliv místnosti dostaneme do kterékoliv jiné. Dále jsem navrhl a implementoval metody pro generaci stromů a řek. Najdeme zde také způsob jak rozdělit svět do několika různých biotopů za pomocí Voroného diagramu. Ve výsledku tu vlastně máme návod, jak vytvořit rozmanitý, bohatý a různorodý svět.

Společně s dalšími vývojáři, pracujícími na tomto projektu, se nám podařilo vytvořit aplikaci, která je v jádru podobná hře Minecraft. Svět je zde také tvořen pomocí bloků, které lze pokrýt libovolnou texturou. Funguje plynulé vykreslování a fyzika. Uživatel se tak může po světě procházet, případně přidávat nebo mazat bloky. Stále je zde ovšem spousty aspektů, které nejsou dotaženy k dokonalosti, ale myslím, že jsme vytvořili dobrý základ pro budoucí vývojáře tohoto projektu.

Během vývoje jsem se snažil o co nejvíce dynamický kód. Ať už se jednalo o jednotlivé bloky, biomy, stromy, budovy, tvory nebo o typy terénů, nic z toho by nemělo být obtížné přidat. Kosmos tak lze velice jednoduše obohatit a rozšířit.

Díky této práci a jejímu tématu, jsem prohloubil své programátorské dovednosti, hlavně co se týče 3D vykreslování. Pochopil a naučil jsem se jak používat perspektivní kameru. Dále jsem více pronikl do praktik procedurálního generování. Navíc se mi povedlo vytvořit procedurální algoritmus pro generaci budov, s jehož výsledkem jsem velice spokojen. S podobným výsledkem jsem uspěl i s generátorem různých typů stromů. Vyzkoušel jsem si tak, co obnáší vytvoření procedurálního algoritmu a také, kolik času a úsilí je pro jeho vytvoření třeba.

Literatura

[1] Minecraft [online]. [cit. 2016-12-19].
Dostupné z: https://minecraft.net/en-us/

[2] BRECHER, Jakub. Bakalářská práce, RPG hra s generováním mapy [online]. [cit. 2017-01-20].
Dostupné z: https://dspace.vsb.cz/bitstream/handle/10084/108878/BRE0084_FEI_B2647_2612R025_2015.pdf?sequence=1&isAllowed=n

[3] GUSTAVSON, Stefan. Simplex noise [online]. [cit. 2017-01-21].
Dostupné z: http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

[4] Procedural Heightmap Terrain Generation [online]. [cit. 2017-02-19].
Dostupné z: https://www.seedofandromeda.com/blogs/58-procedural-heightmap-terrain-generation

[5] Graphical modeling using L-systems [online]. [cit. 2017-03-02].
Dostupné z: http://algorithmicbotany.org/papers/abop/abop-ch1.pdf

[6] ATree3D [online]. [cit. 2017-03-02].
Dostupné z: http://www.avizstudio.com/tools/atree3d/

[7] GrowFX [online]. [cit. 2017-03-02].
Dostupné z: https://exlevel.com/features/

[8] Ridged Perlin Noise [online]. [cit. 2017-04-14].
Dostupné z: http://www.inear.se/2010/04/ridged-perlin-noise

[9] Programujeme Android hru [online]. [cit. 2017-04-15].
Dostupné z: http://www.itnetwork.cz/java/android/programujeme-android-hru/poznavame-libgdx

[10] LibGDX [online]. [cit. 2017-04-15].
Dostupné z: https://libgdx.badlogicgames.com

[11] Gdx-texturepacker [online]. [cit. 2017-04-16].
Dostupné z: https://github.com/libgdx/libgdx/wiki/Project-Setup-Gradle

[12] OpenGameArt [online]. [cit. 2017-04-21].
Dostupné z: https://opengameart.org/

[13] Blender [online]. [cit. 2017-04-21].
Dostupné z: https://www.blender.org/

[14] Fbx-converter [online]. [cit. 2017-04-21].
Dostupné z: http://libgdx.badlogicgames.com/fbx-conv/

[15] The New Yorker [online]. [cit. 2017-04-26].
Dostupné z: http://www.newyorker.com/magazine/2015/05/18/world-without-end-raffi-khatchadourian

[16] Game revolution [online]. [cit. 2017-04-26].
Dostupné z: http://www.gamerevolution.com/features/the-20-most-hideous-no-mans-sky-creatures-youll-ever-see

[17] Planet Minecraft [online]. [cit. 2017-04-26].
Dostupné z: http://www.planetminecraft.com/project/minecraft-frozen---arendelle/

[18] Minecraft Building inc [online]. [cit. 2017-04-26].
Dostupné z: https://minecraftbuildinginc.com/river-of-thyia-custom-terrain/

[19] Game development [online]. [cit. 2017-04-26].
Dostupné z: https://gamedev.stackexchange.com/questions/33590/how-to-generate-caves-that-resemble-those-of-minecraft

[20] Herni snob [online]. [cit. 2017-04-26].
Dostupné z: http://www.hernisnob.cz/2012/06/hra-mesice-rogue-like.html

7 Seznam Příloh

Obsah přiloženého CD:

\src \core - Zdrojové kódy aplikace

\src \desktop - Zdrojové kódy pro PC

\Desktop - Soubor pro spuštění aplikace na PC

\BRE0084.pdf - Diplomová práce ve formátu PDF